Mining Execution Relations for Crosscutting Concerns

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Abstract

Aspect mining tries to identify crosscutting concerns in the code of existing systems and thus supports their adaption to an aspect-oriented design. A semi-automatic static aspect mining approach is described, where the program's control flow graphs are investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in different calling contexts. Two case studies done with the implemented tool show that many discovered candidates for crosscutting concerns are instances of delegation and should not be refactored into aspects. More generally, it is shown that aspect mining techniques need a way to distinguish between delegation and superimposed behaviour.

1 Introduction

Software systems often contain tangled and scattered code. Code is *tangled* if it implements multiple concerns (such as core logic, logging, persistence, security). The notion of *scattered code* refers to code that exists several times in a software system and cannot be encapsulated by separate modules using traditional module systems. The reason for scattering and tangling are most often *crosscutting concerns* (i.e. concerns that crosscut other concerns) that cannot be cleanly separated [1]. The presence of scattered (and tangled) code makes software more difficult to maintain, understand and extend. *Aspect-oriented programming* [2] provides new separation mechanisms for such crosscutting concerns by introducing aspects that can weave code and declarations to specified points in the program.

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and isolate these crosscutting concerns. This task is also called *aspect mining*. Its goal is the detection of crosscutting concerns either for documentation purposes or to enable their refactoring into separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. Aspect mining can also provide insights that enables to classify common aspects which occur in different software systems, such as logging, timing and communication.

It is still an ongoing discussion what types of crosscutting concerns should be refactored into aspects to improve the quality of a system. It is our belief that only superimposed crosscutting behaviour should be refactored, that is, behaviour that is not a system's core functionality (logging and tracing are the prototypical examples of superimposed behaviour). Although the refactoring of other crosscutting concerns is often possible, it will certainly result in a strong coupling between the base and the aspect code, destroying modularity [3] and making the refactored program worser than the original. However, there is not yet enough experience with refactoring of traditional into aspect-oriented programs to judge whether these beliefs are true. Independent of this, identification of the crosscutting concerns in a system is still very useful for the system comprehension.

Several approaches based on program analysis techniques have been proposed for aspect mining (see related work in Section 4 for a discussion). A dynamic program analysis approach [4] that mines aspects based on program traces has been previously developed. During program execution, program traces are generated, which reflect a software system's run-time behaviour. These traces are then investigated for recurring execution patterns. Different constraints

[0]This paper is a postprint of a paper submitted to and accepted for publication in IET Software and is subject to Institution of Engineering and Technology Copyright. The copy of record is available at IET Digital Library. IET Software (2008),2(2):65
http://dx.doi.org/10.1049/iet-sen:20070005

specify when an execution pattern is 'recurring', for example the requirement that the patterns have to exist in different calling contexts in the program trace. The dynamic analysis approach monitors actual (i.e. run-time) program behaviour instead of potential behaviour, as static program analysis does. Because it is not always possible to execute the program that should be analysed, a static analysis variant of the approach [5] that analyses control flow graphs (CFG) for recurring execution patterns has been developed.

This work will first show how the execution relations and their constraints can be generalised to a framework that is applicable to static and dynamic analyses. Then it will present a static analysis to compute execution relations from CFGs, and will present two in-depth case studies of the identified relations in the drawing application JHotDraw and the web application Pet Store. Not surprisingly, the results of the static and dynamic analysis are different because of various reasons such as code not executed, late binding or polymorphism. Because it is well known that the results of a static analysis will differ from its dynamic variant, these differences will not be presented here.

The case studies will show that crosscutting concerns often result from delegation and coding style guides and should not be refactored into aspects. Therefore the evaluation of the results of the static aspect mining approach will specifically try to distinguish crosscutting concerns that are instances of delegation from those that can be regarded as superimposed behaviour. By *delegation* it is meant that an object relies on another to fulfil a delegated task. For example, an object using a list is usually delegating the task to compute the list's size to the list instead of counting the elements of the list (Delegation is used in a very general sense here and not as a programming language feature or in the sense of what is often called *consultation*). In contrast, superimposed behaviour of an object does not belong to the core functionality of the object and there is no other object that directly relies on the result of the superimposed behaviour. An example of superimposed behaviour is if the list, after computing its size, has to notify a logger of the performed computation. This distinction helps in classifying whether a detected crosscutting concern can easily be refactored into aspects by simple refactorings such as those presented by Binkley et al. [6].

The main goal of the paper is to show that aspect mining techniques need a way to distinguish between delegation and superimposed behaviour. It will present a simple filter based on the observation that there is a correspondence between delegation, superimposed behaviour, void and non-void method calls.

The next section contains a description of the static aspect mining approach based on CFGs. Section 3 contains the case studies with the presentation of the filter, followed by related work in Section 4. Section 5 discusses the results and concludes.

2 Aspect mining based on execution relations

The dynamic aspect mining approach previously developed [4] is based on the analysis of program traces, which mirror a system's behaviour in certain program runs. Within these program traces, recurring execution patterns which describe certain behavioural aspects of the software system are identified. It is seen that recurring execution patterns describe recurring functionality in the program and thus are potential crosscutting concerns. To detect these recurring patterns in the program traces, a classification of possible pattern forms based on the so-called *execution relations* has been introduced. They describe in which relation two method executions are in the program trace.

Identifying interesting crosscutting concerns with the dynamic aspect mining approach was possible. However, often enough a program that should be analysed cannot be executed. For such circumstances, a static analysis that extracts the execution relations from CFGs has been developed. The execution relations are a general concept and in the following it is shown how the execution relations and their constraints are defined for this static approach.

A CFG is a directed attributed graph $G = (N, E, n^s, n^e)$ with node set N and edge set E. The program's statements are represented by nodes $n \in N$ and the control flow between statements is represented by *control flow edges* $(n, m) \in E$, written as $n \to m$. E contains control flow edge $n \to m$, iff the statement represented by node m may be executed immediately after the statement represented by n, that is, no other statement is executed in between. Two special nodes $n^s \in N$ and $n^e \in N$ are distinguished: the START node n^s and the EXIT node n^e, which represent beginning and end of the program. Node n^s does not have predecessors and node n^e does not have successors.

Each procedure or method $p \in \mathcal{P}$ of a program is represented with its own CFG $G_p = (N_p, E_p, n_p^s, n_p^e)$, where $\forall p, q : p \neq q \Rightarrow N_p \cap N_q = \emptyset \wedge E_p \cap E_q = \emptyset$ and $N^\star = \bigcup_p N_p$, $E^\star = \bigcup_p E_p$ represent the set of nodes and edges for the complete program that is represented by the graph $G^\star = (N^\star, E^\star)$. Note that a unique EXIT node is assumed, which is a join in the presence of multiple `return` statements (i.e. a `return` statement is a jump to the EXIT node). In languages such as Java, there may be additional multiple exits because of exception handling. Currently, the approach ignores such exits, and execution relations are only identified in the parts of the source code that lead to a normal exit.

Usually, the procedures' or methods' graphs are connected with edges that represent procedure or method calls, however, it is assumed that the called method is available as an attribute of the calling node and therefore such edges are not needed for this paper, and that the graphs are not connected. In the following, only the term 'method' is used to denote methods or procedures.

2.1 Classification of execution relations

The CFGs are focused on method calls (and executions) because imperative or object-oriented systems are to be analysed where logically related functionality is encapsulated in methods.

Crosscutting concerns are reflected by the two different *execution relations* that can be found in CFGs. A method can be executed either after the preceding method execution is terminated or inside of another method's execution. These two cases are distinguished and it is said that there are *outside-* and *inside-execution relations* in CFGs. It is thus defined formally:

$u \rightharpoonup v$ with $u, v \in \mathcal{P}$, is called an *outside-before-execution relation* if there is a path $n_u \rightarrow^\star n_v$ in $G^\star$ where n_u is a call of u, n_v is a call of v and there is no other call on the path. This is read as 'u is executed before v'. $S^{\rightharpoonup}(G)$ is the set of all outside-before-execution relations in a call graph G. This relation can also be reversed, that is $v \leftharpoonup u$ is an *outside-after-execution relation* if $u \rightharpoonup v \in S^{\rightharpoonup}(G)$. The relation $u \leftharpoonup v$ can be read as 'v is executed after u'. The set of all outside-after-execution relations in a graph G is then denoted with $S^{\leftharpoonup}(G)$. fig. 1 shows a small fragment that will be used as an example. The following outside-execution relations can be identified: `a` $\rightharpoonup$ `b`, `b` $\rightharpoonup$ `a`, `a` $\rightharpoonup$ `c`, `b` $\leftharpoonup$ `a`, `a` $\leftharpoonup$ `b` and `c` $\leftharpoonup$ `a`.

$u \in_\top v$ with $u, v \in \mathcal{P}$ is called an *inside-first-execution relation* if there is a path $n_v \rightarrow^\star n_u$ in G_u such that $n_v = n_v^s$ is the START node, n_u is a call of u and there is no other call on the path. The relation $u \in_\top v$ can be read as 'u is executed first in v'. $u \in_\bot v$ is called an *inside-last-execution relation* if there is a path $n_u \rightarrow^\star n_v$ in G_u such that $n_v = n_v^s$ is the EXIT node, n_u is a call of u and there is no other call on the path. The relation $u \in_\bot v$ can be read as 'u is executed last in v'. $S^{\in_\top}(G)$ is the set of all inside-first-execution relations in a CFG G, $S^{\in_\bot}(G)$ is the set of all inside-last-execution relations. In the following, G is dropped when it is clear from the context. In fig. 1, the following inside-execution relations can be identified: `a` $\in_\top$ `m1`, `a` $\in_\top$ `m2`, `a` $\in_\top$ `m3`, `a` $\in_\bot$ `m1`, `a` $\in_\bot$ `m2` and `c` $\in_\bot$ `m3`.

```
void m1() {
  a();
  b();
  a();
}
void m2() {
  a();
  if (...) {
    b();
    a();
  }
}
void m3() {
  a();
  c();
}
```

Figure 1: Example to illustrate execution relations

There is one special case that has to be represented explicitly. Whenever there is a path $n_p^s \rightarrow^\star n_p^e$ that does not contain any call, the following two relations are explicitly generated: $\epsilon \in_\top p$ and $\epsilon \in_\bot p$. These two relations capture the possibility that no call to a method occurs during the execution of method p. Moreover, any inside-first- or inside-last-execution relation also generates similar constraints on the outside execution relations. An inside-first-execution relation $u \in_\top v$ generates $\epsilon \rightharpoonup u$ as there is no other call before u is called and an inside-last-execution relation $u \in_\bot v$ generates $\epsilon \leftharpoonup u$ because there is no other call after u is called. These generated relations are called *epsilon relations*. In fig. 1, the following epsilon relations can

Notation	Relation's name	Read as...	Example
$u \in_\top v$	inside-first	u is executed first in v	$\mathtt{a} \in_\top \mathtt{m3}$
$u \in_\bot v$	inside-last	u is executed last in v	$\mathtt{c} \in_\bot \mathtt{m3}$
$u \rightharpoonup v$	outside-before	u is executed before v	$\mathtt{a} \rightharpoonup \mathtt{c}$
$u \leftharpoonup v$	outside-after	u is executed after in v	$\mathtt{c} \leftharpoonup \mathtt{a}$

Table 1: Summary of the four classes of execution relations

be identified: $\epsilon \rightharpoonup \mathtt{a}$, $\epsilon \leftharpoonup \mathtt{a}$, and $\epsilon \leftharpoonup \mathtt{c}$.

The above defined execution relations are summarised in Table 1 together with an example from fig. 1. They are similar to the execution relations defined for the dynamic aspect mining approach and most of the following constraints can now be used in both approaches.

2.2 Execution relation constraints

Recurring execution relations can be seen as indicators of more general execution patterns. To decide under which circumstances certain execution relations are recurring patterns and thus potential crosscutting concerns in a system, constraints have to be defined. The constraints will also implicitly formalise what crosscutting means.

Formally, an execution relation $s = u \circ v \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoonup, \in_\top, \in_\bot\}$, is called *uniform* if $\forall w \circ v \in S^\circ : u = w$ with $u, v, w \in \mathcal{P} \cup \{\epsilon\}$ holds, that is it exists in always the same composition. $\widehat{U}^\circ$ is the set of execution relations $s \in S^\circ$ which satisfy this requirement. This constraint is easy to explain. Consider an outside-before-execution relation $u \rightharpoonup v$. This is defined as recurring pattern if every execution of v is preceded by an execution of u. The argumentation for outside-after-execution relations is analogous. The uniformity constraint also applies to inside-execution relations. An inside-execution relation $u \in_\top v$ (or $u \in_\bot v$) can only be a recurring pattern if v never executes another method than u as first (or last) method inside its body. In fig. 1, only the following relations are uniform: $\mathtt{a} \rightharpoonup \mathtt{b}$, $\mathtt{a} \rightharpoonup \mathtt{c}$, $\mathtt{a} \leftharpoonup \mathtt{b}$, $\mathtt{a} \in_\top \mathtt{m1}$, $\mathtt{a} \in_\top \mathtt{m2}$, $\mathtt{a} \in_\top \mathtt{m3}$, $\mathtt{a} \in_\bot \mathtt{m1}$, $\mathtt{a} \in_\bot \mathtt{m2}$, $\mathtt{c} \in_\bot \mathtt{m3}$, and $\epsilon \leftharpoonup \mathtt{c}$. In contrast, the relations $\mathtt{c} \leftharpoonup \mathtt{a}$ and $\mathtt{b} \leftharpoonup \mathtt{a}$ are not uniform for example.

It is worth noting that uniformity is easier to achieve for the inside-execution relations because only the method's structure influences the uniformity of the method's inside-execution relations. These relations cannot be uniform only if the method starts or ends with a branch with different method calls. For that reason, all inside-execution relations in fig. 1 are uniform.

The ϵ-relations are now dropped and a further analysis constraint is defined. An execution relation $s = u \circ v \in U^\circ$ with $U^\circ = \widehat{U}^\circ \backslash \{u \circ v \mid u = \epsilon \vee v = \epsilon\}$ is called *crosscutting* if $\exists s' = u \circ w \in U^\circ : w \neq v$ with $u, v, w \in \mathcal{P}$ holds, that is, it occurs in more than a single calling context in the CFG. For inside-execution relations $u \in_\top v$ (or $u \in_\bot v$) the calling context is the surrounding method execution v. For outside-execution relations $u \rightharpoonup v$ (or $u \leftharpoonup v$) the calling context is the method v invoked before (or after) method u. R° is the set of execution relations $s \in U^\circ$ which satisfy this requirement. Execution relations $s \in R^\circ$ reveal *candidates* for crosscutting concerns and aspects as they represent the crosscutting behaviour of the analysed software system. In fig. 1, only the following relations are uniform and crosscutting: $\mathtt{a} \rightharpoonup \mathtt{b}$, $\mathtt{a} \rightharpoonup \mathtt{c}$, $\mathtt{a} \in_\top \mathtt{m1}$, $\mathtt{a} \in_\top \mathtt{m2}$, $\mathtt{a} \in_\top \mathtt{m3}$, $\mathtt{a} \in_\bot \mathtt{m1}$, and $\mathtt{a} \in_\bot \mathtt{m2}$. In contrast, the relation $\mathtt{c} \in_\top \mathtt{m3}$ is uniform but not crosscutting.

The above definitions are exactly the same as in the dynamic approach and the technique to extract the uniform and crosscutting relations (which is not presented here) has been reused in the static approach that is presented next.

2.3 Static aspect mining

A tool has been implemented to extract and evaluate the execution relations from the previous section. This analysis extracts the execution relations from the analysed program's CFGs. In particular, the CFGs are generated for the program's methods and the edges are traversed. The algorithm itself is basically a reaching definitions data flow analysis, where the definitions are replaced by the calls occurring at the CFG's nodes. As this is well known data flow analysis [7], no details are given here and only the basic definitions are given in the following.

Let $C \subseteq N^\star$ be the set of nodes that are calls to a method and let $c(n)$ be the called method of a node $n \in C$. A method p's call at a node n ($p = c(n)$) *reaches* a (not necessarily different) node m, if a path $P = \langle n_1, \ldots, n_k \rangle$ in G exists, such that

1. $k > 1$
2. $n_1 = n \wedge n_k = m$
3. $\forall 1 < i < k : n_i \notin C$

To obey the epsilon relation, START and EXIT nodes are assumed to be call nodes with $\forall p \in \mathcal{P} : n_p^s \in C \wedge n_p^e \in C \wedge c(n_p^s) = \epsilon \wedge c(n_p^e) = \epsilon$.

Now $RC(n)$ is defined as the set of reaching calls at node n. They can be computed via a data flow analysis framework [8]. From the reaching calls sets the four execution relations can be generated:

- $\forall n \in C \wedge m \in RC(n) : m \rightharpoonup n \wedge n \leftharpoonup m$
  Any call that reaches another call causes an outside-before-execution and an outside-after-execution relation between the two calls. (The epsilon relations are automatically generated.)

- $\forall n \in C \wedge n \in N_p \wedge \epsilon \in RC(n) : n \in_\top p$
  If the ϵ call reaches another call, then there exists a path from the START node to the call without another call and the corresponding inside-first-execution relation is generated. (The epsilon relations are generated if the ϵ call reaches the EXIT node.)

- $\forall n_p^e \in C \wedge m \in RC(n_p^e) : m \in_\bot p$
  Any call that reaches the EXIT node generates a corresponding inside-last-execution relation. (The epsilon relation is generated if the ϵ call reaches the EXIT node.)

After the execution relations have been extracted, the constraints are applied and the resulting sets of candidates are presented to the user. He can then decide if a candidate is a crosscutting concern and can be refactored into an aspect. If he decides to refactor a candidate, he can apply the refactorings defined by, for example, Binkley et al. [6]. Each one of the four execution relations' classes corresponds directly to one of refactorings *extract beginning/end of method* and *extract before/after call*.

	number of candidates			
size	$\in_\top$	$\in_\bot$	$\rightharpoonup$	$\leftharpoonup$
2	151	111	85	54
3	56	48	28	16
4	28	12	6	2
5	14	8	7	4
6	9	8	3	3
7	6	5	2	1
8	7	2	1	2
9	3	5	1	
10	2	2	1	2
11	3		2	
12	3	2		
13	3			
14			1	1
18	1			1
19	1			
20		1		1
22	1			
23	1			
25		1		
32		1		
38				1
49	1			
$\sum$	290	206	137	88

Table 2: Number of identified candidates for the four classes of execution relations

3 Experiences

The presented static mining technique has been implemented on top of the Soot framework [9], which is used to compute the analysed program's CFGs. The tool traverses these CFGs and extracts the uniform and crosscutting inside- and outside-execution relations. As a first case, JHotDraw, version 5.4b1, has been analysed, which is a well known system and has often been used in evaluations of aspect mining techniques [10, 11, 12, 13, 14, 6]. It is a drawing application and demonstrates good use of design patterns. It contains about 18.000 lines of source code. Moreover, it has been extensively analysed by Marin et al. [10] and their results are used, which are available on the project's website[1], for comparison. The results will be discussed in detail and the results of a second test case will be presented at this section's end.

[1] `http://swerl.tudelft.nl/amr/`

Table 2 shows the results based on the analysis of 2999 methods that are implemented in the 391 class files found in the distribution. The first column shows the size of detected crosscutting candidates measured by the number of crosscut methods (the number of different methods v for a unique method u with relation $u \circ v \in R^{\circ}$). The following columns show the number of candidates with that size for the four relations inside-first, inside-last, outside-before and outside-after. For example, the last line of Table 2 shows that there is a candidate for a crosscutting concern that crosscuts 49 methods. Overall, the tool has identified[2]

- 290 inside-first candidates from 1090 uniform and crosscutting relations,
- 206 inside-last candidates from 719 uniform and crosscutting relations,
- 137 outside-before candidates from 408 uniform and crosscutting relations and
- 88 outside-after candidates from 335 uniform and crosscutting relations.

It is interesting that there are many more candidates for inside-first or inside-last than for outside-before or outside-after (probably because uniformity is easier to achieve for inside-execution relations as discussed in Section 2.2). Furthermore, there are a lot of candidates with just a small amount of crosscutting, for example, 151 candidates just crosscut two methods as inside-first relation.

Next some of the identified candidates are discussed in detail. However, because of the large number of identified candidates, only the five largest candidates of each category are presented first. This initial discussion is used to argue that many of the identified crosscutting concerns are the result of delegations and should not be refactored into aspects. Instead, a filter is needed that removes the delegations from the results. Such a filter is presented after the first discussion with a detailed presentation of the filtered results.

3.1 Inside-first relations

The largest candidate consists of 49 uniform and crosscutting execution relations. The invoked method is '*...CollectionsFactory.current*'. It is obvious that this is a method to access the current factory object, needed in many other methods of the system. This is clearly crosscutting, however, not a refactorable aspect. This has also been observed by others, for example, Marin et al. [10] classified this method as a utility method which is ignored in their fan-in analysis. Marin et al. defined the set of utility methods by a manual step that may require some familiarity with the subject system. Moreover, they have classified all methods from the Java standard library as utility methods. The same classification will be used in the following to characterise the candidates (but will not be used as a filter).

The second largest candidate consists of 23 relations for the method '*...DecoratedFigure.getDecoratedFigure*'. This is again an accessor method (that returns a figure) and can be regarded as a 'utility' method. Thus, it shows crosscutting behaviour, but is not refactorable.

For the third candidate, things are different. It consists of 22 relations for the method '*...UndoadableAdapter.undo*' that checks whether the current object represents an undo-able action (it just calls the method *isUndoable*). Most calls of the method *undo* have the following form:

```
public boolean undo() {
  if (!super.undo()) {
    return false;
  }
  ...
```

This checks if the superclass already states that this action is not undoable. It is clearly a crosscutting concern and can be refactored into an aspect. However, a few *undo* methods are implemented differently:

```
public boolean undo() {
  if (super.undo()
      && ...additional checks...) {
    ...some cleanup code...
    return true;
  }
  return false;
```

These methods have to be refactored before the crosscutting concern can be extracted.

This candidate belongs to a well known crosscutting concern in JHotDraw, the Undo crosscutting concern which has been refactored by Marin [14]. This

[2]Due to a complete reimplementation with better analysis of exception handling, the reported numbers differ from previously published numbers [8].

refactoring is complicated and not only related to the *undo* method.

The next two candidates consist of 19 and 18 relations for the methods '*...List.size*' and '*...List.iterator*', respectively. Both are utility methods from the Java standard library that return a value which is used at the call sites, thus, they are classified as delegations.

3.2 Inside-last relations

The largest candidate consists of 32 uniform and crosscutting execution relations with the method '*...FigureEnumeration.hasNextFigure*' that is always invoked last inside 32 different methods. Again, it is a utility method that is used to control loops over enumerations. This is basically the same for the next candidate, '*...Iterator.next*' with 25 uniform and crosscutting execution relations.

The third candidate consists of 20 relations for the method '*...List.add*', clearly a utility method. The fourth candidate is another utility method '*...CollectionsFactory.createList*' which is called last in 12 different methods, most of them are constructors.

All of the above candidates have been classified as utility methods by Marin et al. The fifth candidate consists of 12 relations for the method '*...DrawingView.checkDamage*'. This is a concern and has been classified as consistent behaviour concern by Marin et al.

3.3 Outside-before relations

The largest discovered candidate consists of 14 uniform and crosscutting execution relations for the method '*...Iterator.next*'. A closer look at the 14 invocations reveals that this crosscutting is more or less incidental. An operation is performed on a container's next element. It has been classified as a utility method by Marin et al.

The next two largest candidates (consisting of 11 relations) are again more or less incidental crosscutting concerns related to methods '*...DrawingView.drawing*' and '*...List.add*'; they can be regarded as utility methods. The fourth candidate is '*...FigureEnumeration.nextFigure*' with ten relations, also a utility method.

The fifth largest candidate is somewhat interesting. It consists of nine invocations of different methods after a call to '*...AbstractCommand.execute*' (i.e. it is always called before one of the nine other methods is invoked). This method is part of two known concerns in JHotDraw: the Command and the Contract enforcement concern [10]. Eight of the invocations are calls to the '*createUndoActivity*' method of eight different classes. The other is an invocation of '*...ZoomDrawingView.zoomView*', which could be interpreted as an *anomaly*—maybe this deviating behaviour is related to a bug in the program. However, the other eight invocations are of classes representing operations that change the figure and *zoomView* (probably) does not change it, thus this is not an anomaly.

3.4 Outside-after relations

For outside-after execution relations, four of the five largest candidates can clearly be classified as utility methods: '*...FigureEnumeration.hasNextFigure*' has 38 uniform and crosscutting execution relations, '*...Iterator.hasNext*' has 20, '*...List.add*' has 14, and '*...StringBuffer.append*' has 10. The only interesting case is '*...Graphics.setColor*' with 18 relations. Being part of the standard Java library, it has been classified as a utility method by Marin et al. A close look at the preceding method invocations confirms this classification. The crosscutting behaviour is just coincidental.

3.5 A simple filter

It has been seen in the last section that most (17 out of 20) of the examined discovered crosscutting concerns are not to be refactored, because they are perfectly valid in their characteristics; most of them are based on heavy use of delegation to utility methods and accessors. While delegation can often be regarded as crosscutting, crosscutting concerns that are more in the style of superimposition, i.e. that add behaviour at the place where they are used but without having a direct dependence with the enclosing code, need to be identified. A very simple, but very effective, filter is to use the invoked methods' signatures. It is based on the assumption that any method that returns a value has been delegated a part of the calling method's task and the results of the delegated tasks are immediately needed by the delegation method. This is similar to Gybels' and Kellens' [15, 16] unique methods heuristic.

Only void methods are not directly needed where they are invoked. Thus, they represent superimposed

behaviour. The object has to perform a task by invoking the void method, but has no interest in the result itself. Of course, this is over-simplifying because of reference parameters. The non-void methods' return value is used by the calling method, thus, non-void methods are usually delegations from the calling method and not refactorable into aspects. Thus, the filter will remove calls to non-void methods. Moreover, the refactoring of calls to non-void methods to advices is complex, because advices cannot return values.

Sometimes non-void methods are used like void methods and their return value is ignored. A typical example is *List.add*; its return value indicates whether the addition to the list was successful or not—most of the times the programmer knows that it will be successful and ignores the return value. In contrast to the previous experiment [8], non-void methods that are used like a void method are included in the analysis.

This simple filter has two advantages over the utility-method filter used by Marin et al. First, the set of utility methods has to be defined manually for each analysed system. Second, the non-void filter is used during the analysis: nodes that invoke a non-void method are now considered as nodes that do not invoke a method. If there is a path from the entry to a void method invocation, a path from a void method invocation to the exit, or a path from a void method invocation to a second void method invocation, the analysis will generate corresponding relations even if there are non-void invocations on the path. Note that this is different than just applying the filter after the analysis.

The implemented filter extracts only those uniform and crosscutting execution relations that call a void method. The results of this filter are shown in Table 3. The analysis discovers now

- 152 inside-first candidates from 494 uniform and crosscutting relations,
- 141 inside-last candidates from 452 uniform and crosscutting relations,
- 23 outside-before candidates from 73 uniform and crosscutting relations and
- 21 outside-after candidates from 63 uniform and crosscutting relations.

In the following, the extracted crosscutting inside-first-execution relations are discussed for candidates with at least eight relations. A closer look at these relations reveals that many of them have the characteristics of crosscutting concerns, especially the larger ones. Table 4 shows an overview of the results.

	Number of candidates			
size	$\in_\top$	$\in_\bot$	$\rightharpoonup$	$\leftharpoondown$
2	93	85	14	13
3	26	28	3	5
4	11	4	1	
5	5	5	3	2
6	5	8	1	
7	4	2		
8	3	3		
9		4		
10				
11	1		1	
12	1			1
13	1	1		
21	1			
22	1	1		
$\sum$	152	141	23	21

Table 3: Number of candidates when the filter is applied

3.6 Inside-first relations (with filter)

...List.add This method (now called first in 22 different methods) has been discussed in the previous section (in the context of inside-last relations) and has already been classified as a utility method. Note that it is a non-void method used like a void method.

...Graphics.setColor This method has also been discussed before, but in an outside-after relation context. Although it has been classified as a utility method there, a close look to the 21 invoking method reveals that it is now part of a consistent behaviour concern. All invoking methods are drawing methods (15 even have the name *draw*) which need to set the drawing colour first. However, this can be considered as delegation and is better not refactored, because setting the drawing colour is part of drawing's core functionality. Thus, it is (still) better classified as a utility method.

...AbstractTool.mouseDown This method is called first in 13 different instances of the method *mouseDown* and has been already identified by Marin et al. as

part of a consistent behaviour concern. All subclasses of *AbstractTool* that redefine *mouseDown* call its superclass' version first.

...AbstractCommand.execute This method has already been discussed in the previous section and has been classified as part of the command and contract enforcement concerns. Here, it is identified because 12 subclasses of *AbstractCommand* redefine *execute* such that *execute* from the superclass is called first.

...Rectangle.add This method is called first in 11 different methods and belongs to the AWT standard library. It has not been observed by Marin et al. because it has been classified as a utility method. This method reveals a consistent behaviour concern. At the identified call sites a new rectangle is created from two points that have been provided as arguments to the current method:

```
public void basicDisplayBox(Point origin, Point corner) {
  fDisplayBox = new Rectangle(origin);
  fDisplayBox.add(corner);
}
```

It can be seen as a crosscutting concern that could be refactored. However, there exist nine more implementations of *basicDisplayBox* which do not expose the same behaviour.

...Figure.displaybox This is part of a consistent behaviour concern. The invocation is part of eight different methods with the name *invokeStep* that handle eight different directions. It has not been reported by Marin et al., because its fan-in value is below the required threshold.

...Rectangle.translate This method moves a rectangle object by a given vector. It is used eight times in basicMoveBy methods (and there it is most often the only method called):

```
protected void basicMoveBy(int x, int y) {
  fDisplayBox.translate(x,y);
}
```

It is clearly a delegation, but can also be seen as a crosscutting behaviour in the form of a consistent behaviour concern. However, there are nine other implementations of the basicMoveBy method. Five are not using *Rectangle.translate* at all and four use it in a different context. Thus, the *basicMoveBy* methods have no consistent behaviour in respect to *...Rectangle.translate* and this concern is better not refactored into an aspect. This specific concern for the movement of figures has also been identified by Ceccato et al. [11], where it has been discussed if it is 'aspectisable', that is, can be refactored into an aspect. There, three different teams have applied three different aspect mining techniques and the teams were of different opinions.

...ObjectInputStream.defaultReadObject This method is called within eight other methods, all specific versions of *readObject*. It is used to deserialise an object, which is read from an input stream. Thus, this can be classified as a consistent behaviour concern and can be refactored into an aspect. Because this method is part of the standard library, this concern has not been described by Marin et al.

3.7 Inside-last relations (with filter)

The largest candidate is the invocation of '*..List.add*' as the last method in 22 different methods. It is a utility method as discussed before.

The second largest candidate is the invocation of '*...DrawingView.checkDamage*' at 13 different methods. This is a concern as discussed above (and has been classified as consistent behaviour concern by Marin et al.)

The next candidate '*...Figure.displayBox*' is invoked last in nine different methods. This candidate appears also as a inside-first candidate and has been discussed in the previous Section.

The next candidate with nine invocations is '*...Graphics.drawOval*'. It can be regarded as a utility method, because it belongs to the standard library. A closer look at the invocation sites reveals that it belongs to delegation of drawing operations.

Another candidate with nine invocations is '*...StorableOutput.writeStorable*', which is invoked last in nine different *write* methods. This is just a small subset of all its invocations, and it is classified as delegation.

There is one more candidate with nine invocations: '*...Figure.moveBy*' is called last in eight different *moveBy* and one *basicMoveBy* methods. Marin et al. has classified this as belonging to a decorator concern. However, similar to the *Rectangle.translate* discussion

above, it is classified as delegation.

The last three candidates have eight relations: '*...util.add*' is always invoked last in eight different methods. It clearly belongs to the usage of the command design pattern: a command object and a menu shortcut are added to eight different menus. It is clearly a crosscutting command concern, which has not been observed by Marin et al. (because it has been classified as a utility method).

The candidate '*...Graphics.drawRect*' is similar to the above discussed '*...Graphics.drawOval*' and is classified as delegation to a utility method.

The last candidate is '*...StorableOutput.writeInt*', which is also classified as delegation.

3.8 Outside relations (with filter)

There is only one large candidate left after filtering for outside-before execution relations: It is the already discussed candidate '*...AbstractCommand.execute*'. With filtering it is now discovered two more times: it is always invoked before 11 different methods.

For outside-after execution relations, there is also only one large candidate. '*...Undoable.setAffectedFigures*' is always invoked after 12 different methods with the name *setUndoActivity*. Both methods has been classified as utility methods by Marin et al., however, this candidate clearly belongs to the already discussed Undo crosscutting concern.

Table 4 shows an overview of the results. The first column shows the 16 methods that are part of a crosscutting execution relation and have been discussed above. The second column shows the kind of crosscutting execution relation in which the method occurred. The next three columns show if the method has been classified as a utility method (Util.), as delegation (Del.), as a crosscutting concern (Con.) and if this concern has not been observed by Marin et al. (New). If the method is part of a crosscutting concern, the concern's type if given in the last column.

Half of the methods are refactorable crosscutting concerns where four have been previously discarded as utility methods. From the other half, six are classified as utility methods (and thus are implicitly delegations), and two are delegations to methods that are not utility methods. It can be seen that the filter has increased the precision with a small loss of recall, because the *undo* method is no longer identified. It can also be seen that the classification of utility methods has a strong influence on the precision and recall. On the one hand Marin et al.'s classification is too strong, because half of the methods identified as belonging to a concern have been classified as utility method. On the other hand, two of the discussed methods are delegations, but not utility methods.

3.9 Generalisation of discovered patterns

Crosscutting concerns often use method invocations that are similar according to their type or name. For example, Marin et al. [10] not only use the invoked method's static type, but use the set of sub- and superclasses; Tourwe and Mens [17] group by similar identifiers. The approach uses a similar technique to generalise the discovered patterns: The called method $c(n)$ at a node n is identified by its signature either with or without the class it is defined in. For example, two calls *x.add()* and *y.add()*, where x and y have different static types, are either considered two different methods or the same method. This section will discuss the results of the mining if the method's defining class is ignored.

The results of this filter (ignoring the defining class) are shown in Table 5. It is interesting to note that the number of candidates for outside execution relations decrease, whereas the numbers increase for inside execution relations. This is expected and can be considered a feature of the approach. For the outside execution relations, both elements of the relation are affected by the filter, which disturbs uniformity. For the inside execution relations, the filter affects only the left element of the relation, which increases uniformity and crosscutting. Therefore only results for the inside relations are presented next.

The two largest candidates from the inside-first relations are now for the methods '*read*' and '*write*' which are called first inside 26 different methods. These two methods are responsible to read figures from an input stream and write them to an output stream, respectively. Because figures can be composed, the composite figures delegate reading and writing to the embedded figures—it is an instance of the composite design pattern. It is more or less another instance of the consistent behaviour concern, however, not to be regarded as an aspect to be refactored.

The third candidate (22 relations for the method '*add*') and the fourth candidate (21 invocations of '*setColor*') have been discussed above.

Things change for the fifth candidate consisting of 20 candidates for method '*willChange*'. That method

Method	Relation	Util.	Del.	Con.	New	Type
List.add	$\in_\top$	X	X			
Graphics.setColor	$\in_\top$	X	X			
AbstractTool.mouseDown	$\in_\top$			X		consistent behaviour
AbstractCommand.execute	$\in_\top$,$\rightharpoonup$			X		command, contract enforcement
Rectangle.add	$\in_\top$	X		X	X	consistent behaviour
Figure.displaybox	$\in_\top$,$\in_\bot$			X	X	consistent behaviour
Rectangle.translate	$\in_\top$		X			
ObjectInputStream.defaultReadObject	$\in_\top$	X		X	X	consistent behaviour
DrawingView.checkDamage	$\in_\bot$			X		consistent behaviour
Graphics.drawOval	$\in_\bot$	X	X			
StorableOutput.writeStorable	$\in_\bot$	X	X			
Figure.moveBy	$\in_\bot$		X			
util.add	$\in_\bot$	X		X	X	command
Graphics.drawRect	$\in_\bot$	X	X			
StorableOutput.writeInt	$\in_\bot$	X	X			
Undoable.setAffectedFigures	$\leftharpoondown$	X		X	X	undo

Table 4: Overview of the results

	Number of candidates			
size	$\in_\top$	$\in_\bot$	$\rightharpoonup$	$\leftharpoondown$
2	75	76	11	10
3	23	30	3	1
4	9	15	1	4
5	9	7	1	
6	12	11	1	
7	8	5		
8	5	3		
9	2	3		
10	1	4		
11	1			
12	1			
14		1		
15	1	2		
16	1			
19	1			
20	1			
21	1			
22	1	2		
24		1		
26	2			
$\sum$	154	160	17	14

Table 5: Number of candidates with generalisation filter applied

informs a figure that an operation will change the displayed content. This is a crosscutting concern which could be refactored into an aspect. This is a well known crosscutting observer concern in JHotDraw.

The sixth candidate for '*execute*' with 19 invocations has been discussed above. The sixth candidate, '*setUndoable*' is invoked first in 16 methods. This is another method that belongs to the known Undo concern. However, as this is a setter method that just sets a field of the UndoableAdapter class, it is ignored in Marin et al.'s fan-in approach [10].

The rest of the inside-first candidates have all been discussed above and the inside-last candidates will be discussed next.

The largest inside-last candidate is a new candidate: the method '*changed*' is invoked last in 24 different methods. This method is a notifier within the know observer concern in JHotDraw.

The second largest candidate is also not discussed above: '*setRedoable*' is invoked last in 22 methods. It is very similar to the already discussed '*setUndoable*' and belongs to the Undo concern.

The next three candidates have been discussed above: '*read*' (15 invocations), '*checkDamage*' (15) and '*add*' (14).

The next four candidates are all invoked last in ten different methods; '*toolDone*' belongs to a consistent behaviour concern [10], '*repaint*' is used to redraw the

current view and classified as delegation, '*moveBy*' and '*displayBox*' have been discussed above.

3.10 Experiences for the pet store system

As a second test case, Pet Store (version 1.3.2), a sample application for the demonstration of the J2EE platform, has been analysed. The following results are based on an analysis of 1712 methods, 81 methods could not be analysed because Soot was not able to build CFGs for them[3].

Similar to Marin et al. [10], Pet Store has fewer candidates. Overall, the tool has identified (with filter, distinguishing defining classes for invoked methods):

- 64 inside-first candidates from 201 uniform and crosscutting relations,
- 46 inside-last candidates from 161 uniform and crosscutting relations,
- 9 outside-before candidates from 21 uniform and crosscutting relations and
- 12 outside-after candidates from 28 uniform and crosscutting relations.

With filtering and ignoring the defining class:

- 71 inside-first candidates from 266 uniform and crosscutting relations,
- 54 inside-last candidates from 216 uniform and crosscutting relations,
- 7 outside-before candidates from 15 uniform and crosscutting relations and
- 7 outside-after candidates from 17 uniform and crosscutting relations.

An inspection of the identified candidates revealed no refactorable crosscutting concerns. All of the larger candidates are either related to utility methods or are clearly delegations. However, Marin et al. [10] reported four concerns. Three of them are related to exception handling and because exception handling disturbs the linear control flow which is required to mine for recurring execution patters, the approach proposed is not able to discover such concerns. The fourth concern Marin et al. reported is the invocation of the constructor for the *ServiceLocator* class. The approach filters constructor invocations as non-void method invocations, thus this concern has not been discovered. If constructor invocations are included in the analysis, this concern will be discovered, as the constructor invocation for *ServiceLocator* is reported as being invoked first in 17 different methods and invoked before 5 different methods. Then it is the second largest candidate—the largest is the constructor call for *StringBuffer*, which is clearly just a utility invocation.

[3] A planned analysis of the Tomcat system had to be abandoned because we were not able to apply Soot to such a large system.

4 Related work

Aspect mining has been identified as useful technique to understand crosscutting behaviour in non-aspect-oriented programs and as an aid to help in refactoring non-aspect-oriented to aspect-oriented programs. Most of the early approaches were not even semi-automatic because one has to specify a pattern that can be searched for in the source code [18, 19]. Like our approach, more recent approaches (discussed in the following) do not need user specified patterns and identify crosscutting behaviour based on the programs' structure.

The approach most similar to the one proposed is the approach of Marin et al. [10], who use fan-in analysis to identify crosscutting concerns. Fan-in analysis basically counts for each method the number of call sites in the source code that calls the method. This approach is very similar to the inside-first- and inside-last-execution relations; however, the approach proposed is more specific as it only identifies candidates that are easily refactorable by advice.

Gybels and Kellens [15, 16] use heuristics to mine for crosscutting concerns. The 'unique methods' heuristic is defined as 'A unique method is a method without a return value which implements a message implemented by no other method' and can be compared with the non-void methods filter. Gybels and Kellens also search for (unique) methods that are called from many places.

Tourwe and Mens [17] use concept analysis to identify aspectual views in programs. The extraction of elements and attributes from the names of classes, methods and variables, formal concept analysis is used to group those elements into concepts that can be seen as aspect candidates. Tonella and Ceccato [20] also use concept analysis for aspect mining, but they apply it on traces generated by dynamic analysis.

Some other approaches rely on clone detection

techniques to detect scattered code in the form of crosscutting concerns: Bruntink [21, 22, 23] evaluated the use of those clone detection techniques to identify crosscutting concerns. Their evaluation has shown that some of the typical aspects are discovered very well, whereas some are not. Ophir by Shepherd et al. [12] uses a program dependence graph based clone detection technique for aspect mining. After an initial phase that detects clones, a second step filters the candidates and a third phase coalesces the remaining candidates. Candidates identified by the technique or by fan-in analysis often consist of very few code lines; such candidates cannot be identified by clone detection techniques.

Breu and Zimmerman [24] analyse version archives for crosscutting concerns. They consider methods to be part of a crosscutting concern when they are changed together in the same transaction and additionally the changes are the same, that is a call to the same method is inserted. That approach scales to industrial-sized projects discovers cross-cutting concerns across platform-specific code, but needs a version archive of sufficient size and cannot be applied to single-version systems.

Remaining approaches use natural language processing to analyse the identifiers used in source code [25] and clustering of related methods [26].

Ceccato et al. [11] have done a comparison of three aspect mining approaches: fan-in [10], identifier [17] and dynamic analyses [20]. The differences in the approaches and their results are presented and examined. Some of their results have been discussed above.

Timna by Shepherd et al. [13] is a framework for the combination of aspect mining techniques with the goal to increase precision and recall in comparison with approaches that use a single technique. Another framework has been presented by Marin et al. [27], in which they have implemented and combined three aspect mining techniques, one of them is the fan-in analysis mentioned above. The framework identifies a set of requirements to ensure homogeneity in formulating the mining search goals, presenting the results and assessing their quality.

Tonella and Ceccato [28] consider the usage of interfaces as indicators for crosscutting concerns. Implementations of interfaces that belong to one of four criteria are then refactored to an aspect-oriented solution by moving the implementation to an aspect. This approach has been validated by an empirical assessment [29].

Binkley et al. [6] present a semi-automated approach to refactor identified crosscutting concerns in object-oriented programs into aspects. Their refactorings can be used to refactor the identified crosscutting concerns into aspects. Marin [14] has described how the Undo concern in JHotDraw has been refactored into an aspect manually.

Engler et al. [30] use statistical analysis to infer consistent and deviant behaviour based on paired calls that follow one another. The paired calls are very similar to the outside-before and -after execution relations and statistical analysis could be used to find more crosscutting anomalies as presented in the Section 3.3.

5 Discussion, conclusions and future work

This evaluation of the static aspect mining tool has shown that most of the unfiltered identified crosscutting candidates are not concerns refactorable into aspects. This is not much different from results in the previous dynamic aspect mining approach [4]. However, both approaches give interesting insights into some of the crosscutting behaviour of the analysed program. Moreover, the use of a filter that ignores non-void method calls improves the precision of the presented approach.

It is interesting to see a large difference in the reported candidates for two applications that are of similar size (has also been reported by Marin et al. [10]). This suggests that the amount of crosscutting concerns of a certain type is not only dependent on the application's type, but also dependent on the programming style. JHotDraw was developed as a demonstration for good use of design patterns and design patterns are known to have crosscutting concerns.

Based on the previous results from the dynamic approach and the comparison to other mining approaches for the analysed programs, the hypothesis is that aspect mining based on execution relations will have a hard time to identify candidates that are really refactorable into aspects. This hypothesis is in line with other results from similar studies, for example, Marin et al. [10] used a large set of utility methods that are filtered out for JHotDraw. Moreover,

the ongoing refactoring of JHotDraw into a system that makes good use of aspect-oriented programing shows that a refactoring is usually a complex task [14]. In contrast to other authors, it is believed that many detected crosscutting concern in many aspect mining approaches will reveal delegations that should not be refactored into aspects. Delegation can be regarded as a simple form of crosscutting; however, only superimposed tasks that are loosely coupled to the surrounding code can be refactored into aspects.

Therefore future work will continue to develop a filter, which extracts the refactorable candidates from the discovered candidates. The presented simple filter already generates good results. In the end, it is still to be discussed when a refactorable aspect should actually be refactored, similar to the discussion on aspect-oriented program itself [3].

Acknowledgments

The author would like to thank the anonymous reviewers for their feedback on earlier versions of this paper.

References

[1] Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S.M.: 'N Degrees of Separation: Multi-Dimensional Separation of Concerns'. 21st Intl. Conf. on Software Engineering (ICSE). 1999, pp. 107–119

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J.: 'Aspect-Oriented Programming'. European Conf. on Object-Oriented Programming (ECOOP). 1997

[3] Steimann, F.: 'The paradoxical success of aspect-oriented programming'. Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. 2006, pp. 481–497

[4] Breu, S. and Krinke, J.: 'Aspect mining using event traces'. Proc. International Conference on Automated Software Engineering. 2004, pp. 310–315

[5] Krinke, J. and Breu, S.: 'Control-flow-graph-based aspect mining'. Workshop on Aspect Reverse Engineering. 2004

[6] Binkley, D., Ceccato, M., Harman, M., Ricca, F., and Tonella, P.: 'Tool-supported refactoring of existing object-oriented code into aspects'. *IEEE Trans. Software Eng.*, **32** (9), 2006, pp. 698–717

[7] Aho, A.V., Sethi, R., and Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley, 1985

[8] Krinke, J.: 'Mining control flow graphs for crosscutting concerns'. 13th Working Conference on Reverse Engineering: IEEE International Astrenet Aspect Analysis (AAA) Workshop. 2006, pp. 334–342

[9] Vallee-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V.: 'Soot – a java bytecode optimization framework'. Proc. CASCON. 1999

[10] Marin, M., van Deursen, A., and Moonen, L.: 'Identifying crosscutting concerns using fan-in analysis'. *ACM Transactions on Software Engineering and Methodology*, **17** (1), 2007

[11] Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwe, T.: 'Applying and combining three different aspect mining techniques'. *Software Quality Journal*, **14** (3), 2006

[12] Shepherd, D., Gibson, E., and Pollock, L.: 'Design and evaluation of an automated aspect mining tool'. International Conference on Software Engineering and Practice. 2004

[13] Shepherd, D., Palm, J., Pollock, L., and Chu-Carroll, M.: 'Timna: A framework for combining aspect mining analyses'. International Conference on Automated Software Engineering. 2005

[14] Marin, M.: 'Refactoring jhotdraw's undo concern to aspectj'. Workshop on Aspect Reverse Engineering (WARE). 2004

[15] Gybels, K. and Kellens, A.: 'An experiment in using inductive logic programming to uncover pointcuts'. First European Interactive Workshop on Aspects in Software. 2004

[16] Gybels, K. and Kellens, A.: 'Experiences with identifying aspects in smalltalk using 'unique methods". Workshop on Linking Aspect Technology and Evolution (LATE). 2005

[17] Tourwe, T. and Mens, K.: 'Mining aspectual views using formal concept analysis'. Proc. IEEE International Workshop on Source Code Analysis and Manipulation. 2004

[18] Griswold, W.G., Kato, Y., and Yuan, J.J.: 'Aspect Browser: Tool Support for Managing Dispersed Aspects'. Technical Report CS99-0640, Department of Computer Science and Engineering, UC, San Diego, 1999

[19] Zhang, C. and Jacobsen, H.A.: 'Quantifying Aspects in Middleware Platforms'. 2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD). 2003, pp. 130–139

[20] Tonella, P. and Ceccato, M.: 'Aspect mining through the formal concept analysis of execution traces'. 11th IEEE Working Conference on Reverse Engineering (WCRE 2004). 2004

[21] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwe, T.: 'An evaluation of clone detection techniques for identifying cross-cutting concerns'. Proc. International Conference on Software Maintenance. 2004

[22] Bruntink, M.: 'Aspect mining using clone class metrics'. Workshop on Aspect Reverse Engineering. 2004

[23] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwe, T.: 'On the use of clone detection for identifying crosscutting concern code'. *IEEE Trans. Software Eng.*, **31** (10), 2005, pp. 804–818

[24] Breu, S. and Zimmermann, T.: 'Mining aspects from version history'. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). 2006, pp. 221–230

[25] Shepherd, D., Tourwe, T., and Pollock, L.: 'Using language clues to discover crosscutting concerns'. First International Workshop on the Modeling and Analysis of Concerns in Software (MACS). 2005

[26] Shepherd, D. and Pollock, L.: 'Interfaces, aspects, and views'. Workshop on Linking Aspect Technology and Evolution (LATE). 2005

[27] Marin, M., Moonen, L., and van Deursen, A.: 'A common framework for aspect mining based on crosscutting concern sorts'. Proceedings of the 13th IEEE Working Conference on Reverse Engineering (WCRE). 2006, pp. 29–38

[28] Tonella, P. and Ceccato, M.: 'Migrating interface implementation to aspects'. 20th IEEE International Conference on Software Maintenance (ICSM'04). 2004, pp. 220–229

[29] Tonella, P. and Ceccato, M.: 'Refactoring the aspectizable interfaces: An empirical assessment'. *IEEE Trans. Software Eng.*, **31** (10), 2005, pp. 819–832

[30] Engler, D., Chen, D.Y., Hallem, S., Chou, A., and Chelf, B.: 'Bugs as deviant behavior: a general approach to inferring errors in systems code'. *SIGOPS Oper. Syst. Rev.*, **35** (5), 2001, pp. 57–72