

# A Qualitative Comparison of Three Aspect Mining Techniques

M. Ceccato<sup>(1)</sup>, M. Marin<sup>(2)</sup>, K. Mens<sup>(3)</sup>, L. Moonen<sup>(2,4)</sup>, P. Tonella<sup>(1)</sup>, T. Tourwé<sup>(4)</sup>

<sup>(1)</sup> ITC-irst, Trento, Italy

<sup>(2)</sup> Delft University, The Netherlands

<sup>(3)</sup> Université catholique de Louvain, Belgium

<sup>(4)</sup> CWI, The Netherlands

ceccato@itc.it, a.m.marin@ewi.tudelft.nl, kim.mens@info.ucl.ac.be,  
leon.moonen@computer.org, tonella@itc.it, tom.tourwe@cwil.nl

## Abstract

*The fact that crosscutting concerns (aspects) cannot be well modularized in object oriented software is an impediment to program comprehension: the implementation of a concern is typically scattered over many locations and tangled with the implementation of other concerns, resulting in a system that is hard to explore and understand. Aspect mining aims to identify crosscutting concerns in a system, thereby improving the system's comprehensibility and enabling migration of existing (object-oriented) programs to aspect-oriented ones. In this paper, we compare three aspect mining techniques that were developed independently by different research teams: fan-in analysis, identifier analysis and dynamic analysis. We apply each technique to the same case (JHotDraw) and mutually compare the individual results of each technique based on the discovered aspects and on the level of detail and quality of those aspects. Strengths, weaknesses and underlying assumptions of each technique are discussed, as well as their complementarity. We conclude with a discussion of possible ways to combine the techniques in order to achieve a better overall aspect-mining technique.*

**Keywords:** crosscutting concerns, aspect-oriented programming, aspect mining, fan-in analysis, concept analysis.

## 1. Introduction

The tyranny of the dominant decomposition [11] states that no matter how well a system is decomposed into modular units like functions and classes, some functionality will always cut across that modularity. Some well-known examples of these *crosscutting concerns* include persistence, session management, logging and error handling.

From a program comprehension point of view, crosscutting concerns are a burden for two reasons. First of all, discovering or understanding the implementation of a specific concern is difficult, as the concern is not localised in one single module but *scattered* over many different modules. Secondly, understanding the implementation of the modules

themselves becomes more difficult, since the code of the different concerns is *tangled* with the main functionality of those modules.

*Aspect mining* tools try to identify such concerns semi-automatically. Semi-automated identification of concern code is an absolute necessity given the size and complexity of current-day software systems. It allows a bottom-up, divide-and-conquer, comprehension strategy where developers can study and understand each concern in isolation, without worrying about the other code [1]. Moreover, it (eventually) allows developers to refactor the system into an aspect-oriented one, where some or all crosscutting concerns are cleanly captured inside so-called *aspects*. Aspect mining is thus an important prerequisite step for aspect refactoring, but is also valuable as a software exploration technique by itself [9].

In this paper, we compare three aspect mining techniques that were developed independently: fan-in analysis [7], identifier analysis [8, 14] and dynamic analysis [13]. We discuss the strengths, weaknesses and underlying assumptions of each of them. Our goal is not to decide which technique is ‘best’ at identifying relevant aspects, because it is hard (if not impossible) to define the optimal modularization into classes and aspects for a given system. Different design decisions on what is an aspect and what is part of the principal decomposition may be equally good, thus it is hard to come out with a commonly agreed set of *relevant* aspects. Rather, we aim at finding out how the independent techniques complement each other and can be combined so as to minimise their weaknesses and maximise their strengths. The lack of a clearly definable benchmark, to be used as the reference for comparison, prevented us from conducting a *quantitative* study and constrained us to consider a set of *qualitative* comparison criteria.

Our contributions can be summarized as follows:

- we explain what the three aspect mining techniques are and are not capable of, which is useful knowledge for software engineers interested in using aspect mining;

- we present a detailed comparison of the different techniques and discuss how they can be combined in order to achieve better results. Such information is also relevant to other researchers working on aspect mining to get a better understanding of how their individual techniques compare to and could be combined with ours;
- the results of applying the techniques on a common benchmark application, JHotDraw 5.4b, document many of the concerns present in that application. This is of particular interest to JHotDraw developers and users, but also to other aspect mining researchers. JHotDraw may well become the de facto benchmark for aspect mining;

The remainder of this paper is structured as follows. In Section 2 we introduce the necessary background concepts to understand the three aspect mining techniques that are explained in Section 3, followed by a discussion of their individual results in Section 4. In Section 5 we compare these individual results and in Section 6 we draw conclusions on how to combine the techniques so as to obtain a better overall aspect mining technique. Section 7 concludes the paper. For an overview of related work concerning aspect mining, we refer to the papers discussing the individual techniques [1, 7, 8, 13, 14].

## 2. Background concepts

### 2.1. Fan-in

The *fan-in* metric, as defined by Henderson-Sellers, counts the number of locations from which control is passed into a module [6]. In the context of object-orientation, the module-type to which this metric is applied is the method. We define the *fan-in* of a method  $M$  as the number of distinct method bodies that can invoke  $M$ . Because of polymorphism, one call site can affect the fan-in of several methods: a call to method  $M$  contributes to the fan-in of  $M$ , *but also* to all methods refined by  $M$ , *as well as* to all methods that are refining  $M$ .

This interpretation of the fan-in metric in the context of polymorphism corresponds to the standard behavior of the *search for references* feature of the Eclipse IDE, which provides an easy way of implementing the metric.

### 2.2. Concept analysis

Formal concept analysis (FCA) [4] is a branch of lattice theory that can be used to identify meaningful groupings of *elements* that have common *properties*.<sup>1</sup> FCA takes as input a so-called *context*, which consists of a (potentially large, but

<sup>1</sup>We use the terms *element* and *property* instead of *object* and *attribute* used in traditional FCA literature, because these latter terms have a very specific meaning in OO software development.

finite) set of *elements* and a set of *properties* on those elements. Starting from such a context, FCA determines *maximal* groups of elements and properties, called *concepts*, such that each element of the group shares the properties, every property of the group holds for all of its elements, no other element outside the group has those same properties, nor does any property outside the group hold for all elements in the group.

The containment relationship between these groups of elements and properties defines a partial order over the set of all concepts, which can be shown to be a lattice [4]. The lattice's bottom concept contains those elements that have all properties. The top concept contains those properties that hold for all elements. A concept is a sub-concept of another one, its super-concept, if its properties are a superset of the super-concept's properties and its elements are a subset of the super-concept's elements. Intuitively, the sub-concept relationship can be interpreted as a specialisation of more general notions: the lower we are in the lattice the more specific the concepts become (more properties to satisfy and less elements that satisfy them).

The nodes (concepts) of a concept lattice are typically labelled with their full set of associated properties and elements. With *sparse labeling* of the lattice, however, we label a node with an element only if it is the most specific (i.e., lowest) node having that element, and we label a node with a property only if it is the most general (i.e., highest) node having that property. This labelling scheme is much more compact without loss of information.

## 3. The three aspect mining techniques

In this section, we give a brief overview of each of the three techniques, developed independently by different research groups, that support the semi-automated discovery of potential aspects in the source code of an object-oriented software system that was written in a non aspect-oriented way.

### 3.1. Fan-in analysis

Based on earlier studies of crosscutting concerns described in literature, we observed that crosscutting functionality can occur at different levels of modularity. Classes, for instance, can assimilate new concerns by implementing multiple interfaces or by implementing new methods specific to superimposed roles. At the method level, crosscutting in many cases resides in calls to methods that address a different concern than the core logic of the caller. Typical examples include logging, tracing, pre- and post-condition checks, and exception handling. It is exactly this type of crosscutting that we capture with fan-in analysis.

When we study the mechanics of AOP, we see that it employs the so-called *advice* construct to eliminate crosscutting at the method level. This construct is used to acquire control

of program execution and add crosscutting functionality to methods without an explicit invocation from that method. As a result, it isolates the crosscutting nature of these concerns in a separate module.

In our approach, we reverse this line of reasoning and employ *fan-in* analysis in the system’s source code to find symptoms of code scattering. In this case, concerns present themselves as a number of distributed calls to a method implementing a crosscutting functionality and the *amount* of calls (fan-in) is a good measure for the importance and scattering of the discovered concern.

To perform the fan-in analysis, we implemented the metric as a plug-in for the Eclipse platform, and integrated it into an iterative process that consists of three steps:

1. Automatic computation of the fan-in metric for all methods in the investigated system.
2. Filtering of the results from the previous step by
  - eliminating all methods with fan-in values below a chosen threshold (in the experiment, we used a threshold of 10);
  - eliminating the accessor methods (methods whose signature matches a *get\*/set\** pattern and whose implementation only returns or sets a reference);
  - eliminating utility methods, like *toString()* and collection manipulation methods, from the remaining subset.
3. (Largely manual) analysis of the methods in the resulting, filtered set by exploring the callers and call sites, the naming conventions used, the implementation and the comments in the source code.

The last step is partly supported by the implemented plug-in through reports of the caller lists for each investigated method. The callers can be organized per package or class, thus facilitating the inspection of the calling context.

### 3.2. Identifier analysis

In the absence of designated language constructs for aspects, naming conventions are the primary means for programmers to associate related but distant program entities. This is especially the case for object-oriented programming, where polymorphism allows methods belonging to different classes to have the same signature, where it is good practice to use intention-revealing names, and where design and other programming patterns provide a common vocabulary known by many programmers.

*Identifier analysis* relies on this assumption and tries to identify potential aspects and crosscutting concerns by grouping program entities with similar names. More specifically, it applies FCA with as elements all classes and methods in the analysed program (except those that generate too

much noise in the results, like test classes and accessor methods), and as properties the identifiers associated with those classes and methods.

The identifiers associated to a method or class are computed by splitting up its name based on where capitals appear in it. For example, a method named `createUndoActivity` yields three identifiers `create`, `undo` and `activity`. In addition, we apply the Porter stemming algorithm [10] to make sure that identifiers with the same root form (like `undo` and `undoable`) are mapped to one single representative identifier or ‘stem’. It is these stems that are used as properties for the concept analysis.

The FCA algorithm then groups entities with the same identifiers. When such a group contains a certain minimum number of elements (in the experiment, a threshold of 4 was used) the group is considered a seed for a potential aspect. The only remaining but most difficult task is that of deciding manually whether a concept identifies a valid aspect. To help the developer in this last task, our *DelFSTof* source-code mining tool presents the concepts in such a way that they can be browsed easily by a software engineer and so that he or she can readily access the code of the classes and methods belonging to a discovered concept.

### 3.3. Dynamic analysis

Formal concept analysis has been used to locate ‘features’ in procedural programs [2]. In that work, the goal was to identify the computational units (procedures) that specifically implement a feature (i.e., requirement) of interest. Execution traces obtained by running the program under given scenarios provided the input data (dynamic analysis).

In a similar way, dynamic analysis can be used to locate aspects in program code [13] according to the following procedure. Execution traces are obtained by running an instrumented version of the program under analysis, for a set of scenarios (use-cases). The relationship between execution traces and executed computational units (methods) is subjected to concept analysis. The execution traces associated with the use-cases are the elements of the concept analysis context, while the executed methods are the properties. In the resulting concept lattice (with ‘sparse labeling’), the *use-case specific* concepts are those labeled by at least one trace for some use-case (i.e. the concept contains at least one specific property) while the concepts with zero or more properties as labels are regarded as *generic* concepts. Thus, use-case specific concepts are a subset of the generic ones.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios. When the methods that label one such concept (using the ‘sparse labeling’) crosscut the principal decomposition, a candidate aspect is determined. More specifically, a concept is a candidate aspect if: (1) *scatter-*

*ing*: more than one class contributes to the functionality associated with the given concept (i.e., the methods labeling the concept belong to more than one class); (2) *tangling*: the class itself addresses more than one concern (i.e., appears in more than one use-case specific concept).

The first condition alone is typically not sufficient to identify crosscutting concerns, since it is possible that a given functionality is allocated to several modularized units without being tangled with other functionalities. In fact, it might be decomposed into sub-functionalities, each assigned to a distinct module. It is only when the modules specifically involved in a functionality contribute to other functionalities as well that crosscutting is detected, hinting for a candidate aspect.

### 3.4. Tangling and scattering

As the two main indicators of aspects in program code are *scattering* and *tangling*, we briefly discuss to what extent each of the techniques described look for these indicators when mining for aspects.

Both fan-in analysis and identifier analysis focus on detecting scattering, albeit in different ways: fan-in analysis identifies potential aspects by looking for scattered calls, while identifier analysis looks for scattering of similar identifiers. They do not explicitly consider tangling, however. Dynamic analysis, on the other hand, tries to find code that is tangled as well as scattered: it looks for use-case specific methods that are scattered over different classes, and in addition requires that there is some tangling of multiple functionalities in the involved classes.

Nevertheless, despite the fact that they do not explicitly address tangling, fan-in analysis and identifier analysis are able to detect quite some interesting aspects and crosscutting concerns. One reason for this is that not all concerns are necessarily tangled. A second reason is that the techniques do consider tangling implicitly. For example, when a single method calls many different other methods it contributes to the fan-in value of many different other methods. The individual method itself thus addresses different concerns, which can be considered as tangling. Similarly, the fact that a method addresses different concerns is often reflected in the identifiers chosen for that method.

Although both techniques could be extended to consider tangling more explicitly, in Section 6, we discuss rather how a combination of the different techniques might lead to an aspect mining technique that performs even better.

## 4. Results of the aspect mining

In this section, we present the results of applying each technique to version 5.4b1 of JHotDraw, a Java program with approximately 18,000 non-commented lines of code and around 2800 methods. JHotDraw is a framework for drawing

structured 2D graphics and was originally developed as an exercise to illustrate good use of object-oriented design patterns [3] in a Java program. These particularities recommend it as a well-designed case study, a prerequisite for considering improvements through aspect-oriented techniques. Furthermore, it also shows that modularisation limitations are present in even well-designed (legacy) systems.

### 4.1. The fan-in analysis experiment

As described in Subsection 3.1, fan-in analysis first performs a number of successive steps to filter the methods in the analyzed system. The threshold-based filtering, which selects methods with high fan-in values, kept around 7% of the total number of methods. The filters for accessors and utility methods eliminated around half of the remaining methods. In the remaining subset, more than half of the methods (52%) were categorized as *aspect seeds*: methods that by the structure of the calls and the call sites strongly connote a crosscutting concern and thus offer the core elements to further explore and understand the whole extent of the concern's implementation. This seed/non-seed categorization was largely based on manual analysis. It identified the types of crosscutting concerns presented in Table 1. Several of these concern types, such as *consistent behavior* or *contract enforcement* [12], have more than one instance in JHotDraw; that is, multiple unrelated (crosscutting) concerns exist that conform to the same general description. For example, one instance of *contract enforcement* checks a priori conditions to a command's execution, while another instance verifies common requirements for activating drawing tools. The number of different instances that were detected is indicated in the # column.

We distinguish three situations in which the fan-in metric can be associated to the crosscutting structure of a concern implementation:

1. The crosscutting functionality is implemented through a method and the crosscutting behavior resides in the explicit calls to this method. Examples in this category include *consistent behavior* and *contract enforcement*.
2. The implementation of the crosscutting concern is scattered throughout the system, but makes use of a common functionality. The crosscutting resides in the call sites, and can be detected by looking at the similarities between the calling contexts and/or the callers. Examples of concerns in this category are *persistence* and *undo*.
3. The methods reported by the fan-in analysis are part of the roles superimposed to classes that participate in the implementation of a design pattern. Many of these roles have specific methods associated to them: the *subject*

Concern type	#	Seed's description
Consistent behavior	4	Methods implementing the consistent behavior shared by different callers, such as checking and refreshing figures that have been changed when executing a command.
Contract enforcement	4	Method implementing a contract that needs to be enforced, such as checking the reference to the editor's active view before executing a command.
Undo	1	Methods checking whether a command is undoable/redoable and the <i>undo</i> method in the superclass which is invoked from the overriding methods in subclasses.
Persistence and resurrection	1	Methods implementing functionality common to persistent elements, such as read/write operations for primitive types wrappers (e.g., Double, Integer, etc.) which are referenced by the scattered implementations of persistence/resurrection.
Command design pattern	1	The <i>execute</i> method in the command classes and command constructors.
Observer design pattern	1	The observers' manipulation methods and <i>notify</i> methods in classes acting as subject.
Composite design pattern	2	The composite's methods for manipulating child components, such as adding a new child.
Decorator design pattern	1	Methods in the decorator that pass the calls on to the decorated components.
Adapter design pattern	1	Methods that manipulate the reference from the adapter( <i>Handle</i> ) to the adaptee( <i>Figure</i> ).

**Table 1. Summary of the results of the fan-in analysis experiment.**

role in an Observer design pattern is responsible to notify and manage the observer objects, while the *composite* role has manipulation methods as specific children. In general, establishing a relation between these seed-methods and the complete concern to which they appertain might require a better familiarity of the human analyzer with the code being explored, than for the previous two categories. However, many of these patterns are well-known and have a clear defined structure, which eases their recognition [5].

Table 1 details the types of crosscutting concerns that were identified and the seeds that led to their identification. For more details we refer to [7].

#### 4.2. The identifier analysis experiment

Applying the identifier analysis technique of Subsection 3.2 on JHotDraw yielded 230 concepts and took about 31 seconds when using a threshold of 4 for the minimum number of elements in a concept. With a threshold of 10, the number of concepts produced was significantly less: only 100 concepts remained after filtering, for a similar execution time. In both cases, 2193 elements and 507 properties were considered. It is a good sign that the number of properties is significantly smaller than the total number of elements considered, as it implies that there is quite some overlap in the identifiers of the different source-code entities, which was one of our premisses.

Table 2 presents some of the candidate aspects and crosscutting concerns discovered by manually analyzing the classes and methods belonging to the extent of the concepts produced by the FCA algorithm. The first column names the concern, the second column shows the identifiers shared by the elements belonging to the concept(s) corresponding to that concern. When multiple concepts (identifiers) correspond to one single concern, they are separated by a '/'. The

```
CH.ifa.draw.figures:
  EllipseFigure.basicMoveBy(int,int)
  PolyLineFigure.basicMoveBy(int,int)
  RectangleFigure.basicMoveBy(int,int)
  RoundedRectangleFigure.basicMoveBy(int,int)
  TextFigure.moveBy(int,int)
CH.ifa.draw.standard:
  AbstractFigure.moveBy(int,int)
  DecoratorFigure.moveBy(int,int)
```

**Figure 1. Specific methods corresponding to the discovered aspect *Move figure*.**

third column shows the size of the extent for each concept. Finally, for illustration purposes, the fourth column shows some program entities appearing in the extent of the discovered concepts.

We retained 41 crosscutting concerns out of 230 concepts, when we used a threshold of 4 for the minimum number of elements in a concept. We categorized these discovered concerns in three different categories. (1) Some of these concerns looked like aspects in the more traditional sense (e.g., *observer*, *undo* and *persistence*). (2) Many other concerns seemed to represent a crosscutting functionality that was part of the business logic (e.g., *drawing figures*, *moving figures*). The distinction between these two first categories is rather subjective, however. (3) We also discovered three Java-specific concerns (e.g., *iterating over collections*) that are difficult to factor out into an aspect because they rely on or extend specific Java code libraries.

Crosscutting concern	Concept(s)	#elements	Some elements
Observer	change(d) / check / listener / release	67 / 14 / 65 / 12	figureChanged(e) / checkDamage() / createDesktopListener() / ...
Undo	undo(able) / redo(able)	53 / 14	createUndoActivity() / redo()
Visitor	visit	12	visit(FigureVisitor)
Persistence	file / storable / load / register	15 / 5 / 8 / 7	registerFileFilters(c) / readStorable() / loadRegisteredImages
Drawing figures	draw	112	draw(g)
Moving figures	move	36	moveBy(x,y), moveSelection(dx,dy)
Iterating over collections	iterator	5	iterator(), listIterator(), ...

**Table 2. Selection of results of the identifier analysis experiment.**

### 4.3. The Dynamic Analysis experiment

The dynamic analysis technique of Subsection 3.3 is supported by the *Dynamo* aspect mining tool<sup>2</sup>. The first step required by *Dynamo* is the definition of a set of use-cases. To accomplish this task we used the documentation associated with the main functionalities of JHotDraw and defined a use-case for each functionality described in the documentation. For example, we created a use-case to draw a rectangle, one to draw a line using the scribble tool, one to create a connector between two existing figures, one to attach a URL to a graphical element, and so on. In total we obtained 27 use-cases. When executed they exercised 1262 methods belonging to JHotDraw classes, so that the initial context for the concept analysis algorithm contained 27 elements and 1262 properties. The resulting concept lattice contained 1514 nodes.

Of all the concepts in the lattice, based on the crosscutting conditions of scattering and tangling, 11 were classified as use-case specific aspects, while 56 (including those 11) were considered as generic aspects. We then revisited both the use-case specific and generic concepts manually, in order to determine which ones could be regarded as plausible aspects and which ones should be considered false positives. The criterion we followed in this assessment was the following: a concept satisfying the crosscutting conditions is considered a candidate aspect if

- it can be associated to a single, well-identified functionality (this usually accounts for the possibility to give it a short description that labels it), and
- some of the classes involved in such a functionality have a different primary responsibility (indicating crosscutting with respect to the principal decomposition).

Of course, due to the nature of aspects and the related design decisions, some level of subjectivity still remains (as is the case for the other techniques).

<sup>2</sup>Available from <http://star.itc.it/dynamo/> under GPL.

Aspect	Concepts	Methods
Undo	2	36
Bring to front	1	3
Send to back	1	3
Connect text	1	18
Persistence	1	30
Manage handles	4	60
Move figure	1	7
Command executability	1	25
Connect figures	1	55
Figure observer	4	11
Add text	1	26
Add URL to figure	1	10
Manage figures outside drawing	1	2
Get attribute	1	2
Set attribute	1	2
Manage view rectangle	1	2
Visitor	1	6

**Table 3. Summary of the results of the dynamic analysis experiment.**

Figure 1 shows an example of a concept that was classified as a candidate aspect. In addition to satisfying the crosscutting conditions, the methods labeling the concept in the sparse representation are associated to a clearly identified functionality (*Move figure*) and the involved classes have another primary responsibility (grouping features of drawable figures).

In the end, the list of candidate aspects shown in Table 3 was obtained. The four topmost aspects are use-case specific. As apparent from the second column of the table, and as was the case for the identifier analysis experiment, some aspects were detected by multiple concepts. In total, among the 56 generic concepts satisfying the crosscutting conditions, 24 concepts were judged to be associated with 18 candidate aspect.

The methods associated with each candidate aspect (counted in the last column of Table 3, see also Figure 1) are indicative of the “aspectizable” functionality. Although

they may be not the complete list (dynamic analysis is partial) and may contain false positives, they represent a good starting point (“seeds”) for a refactoring intervention aimed at migrating the application to AOP.

## 5. Interpretation of the results

In this section we discuss some selected concerns that were identified by the different techniques. We selected a concern that was detected by all three techniques, as well as a representative set of concerns that were detected by some techniques but not by others. This allows us to clearly pinpoint the strengths and weaknesses of each individual technique, which in turn enables us to propose different ways of combining the techniques to achieve better results.

### 5.1. Selected concerns

Table 4 summarizes the concerns we selected. The first column names the concern. The other columns show by what technique(s) the concern was discovered: if a technique discovered the concern, we put a + sign in the corresponding column, otherwise a - sign is in the table.

**Observer** The Observer design pattern is an example of a concern reported by all techniques. Other examples include *undo* functionality and *persistence*, whose implementation in JHotDraw is described in [7]. Their identification should come as no surprise, because they correspond to well-known aspects, frequently mentioned in AOP literature, or to functionalities for which an AOP implementation looks quite natural.

Concerns identified by all three techniques are probably the best starting point for migrating the given application to AOP, because developers can be quite confident that the concern is very likely to be an aspect. However, the fact that only four of such aspects were discovered, stresses the need for an approach that combines the strengths of different techniques.

**Contract enforcement / consistent behavior** The *contract enforcement* and *consistent behavior* concerns [12] generally describe common functionality required from, or imposed on, the participants in a given context, such as a specific pre-condition check on certain methods in a class hierarchy. An example from the JHotDraw case 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).

We classify these concerns as combination of contract enforcement and consistent behavior since these types often have very similar implementations and choosing for a particular type depends mainly on the context and (personal) interpretation.

Fan-in analysis is particularly suited to address this kind of scattered, crosscutting functionalities, which involve a large number of calls to the same method, while the other two

techniques potentially miss it. In fact, contract enforcement and consistent behavior are usually associated with method calls that occur in *every* execution scenario, so that they cannot be discriminated by any specific use-case. On the other hand, identifier analysis will miss the likely cases were the methods that enforce a given contract or ensure consistent behavior do not share a common naming scheme.

**Command execution** Revisiting the example of the *execute* methods in the *Command* hierarchy, we observe that the identifier analysis technique identified a concept with exactly these methods. Indeed, all *execute* methods have the same name and manual inspection showed they exhibit similar behavior: they nearly all make a super call to an *execute* method, invoke a *checkDamage* method and (though not always) invoke a *setUndoActivity* and *getUndoActivity* method.

Hence, whereas identifier analysis may not detect the more generic contract enforcement / consistent behavior aspect directly, it can identify some locations (pointcuts) where potentially such an aspect could be introduced. Of course, the technique is currently rather lightweight and only identifies locations that share similar identifiers. However, by extending the approach so that it takes similarities in the method bodies into account rather than similarities in the method signatures alone, we expect a decrease in the number of false positives and false negatives.

**Bring to front / Send to back** The functionality associated with this concern consists of the possibility to bring figures to the front or send them to the back of an image. When exercised, it executes specific methods that have a low fan-in, hence they were not detected by fan-in analysis. Identifier analysis also missed them, because there were not enough methods with a sufficiently similar name to surpass the threshold. Hence, dynamic analysis is the only technique that identified this concern. This example is a good representative of crosscutting concerns that are reported only by dynamic analysis: whenever the methods involved in a functionality are not characterized by a unifying naming scheme (or there are not enough of them), neither do they have high fan-in, the other two techniques are likely to fail.

**Manage handles** A crosscutting functionality is responsible for managing the handles associated with the graphical elements. Such handles support interactive operations, such as resizing of an element, conducted by clicking on the handle and dragging the mouse. This candidate aspect is interesting because it is detected by dynamic analysis and by identifier analysis, but in different ways. Identifier analysis detects this concern based on the presence of the word ‘handle’ in identifiers. Consequently, it misses methods such as *north()*, *south()*, *east()*, *west()*, which are clearly related to this concern, but do not share the lexicon with the others. On the other hand, dynamic analysis reports both the latter methods and (some of) those containing the word ‘handle’. However, since not all possible handle interactions have been

Concern	Fan-In Analysis	Identifier Analysis	Dynamic Analysis
Observer	+	+	+
Consistent behavior / Contract enforcement	+	-	-
Command execution	+	+	-
Bring to front / Send to back	-	-	+
Manage handles	-	+	+
Move Figures	+	+	+

**Table 4. A selection of detected concerns in JHotDraw.**

exercised, the output of dynamic analysis is partial and does not include all the methods reported by identifier analysis.

The *manage handles* concern was missed by the fan-in analysis because the calls are too specific: they are similar but different calls instead of one single called method with a high fan-in.

**Moving figures** The three techniques discard concerns on different bases: some of the concerns are filtered automatically while others are excluded manually. The *move figures* concern, seeded by the *moveBy* method in the *Figure* classes, is one example where different, subjective decisions can be made depending on whether the concept is classified either as a candidate aspect or as part of the principal decomposition. The *moveBy* methods allow to move a figure with a given offset. The team who used fan-in analysis argued that the original design seems to consider this functionality as part of a *Figure*'s core logic. The other two teams considered it as part of a crosscutting functionality and included it in the list of reported candidates.

This example highlights the difficulty of deciding objectively on what is and what is not an aspect and corroborates our choice to conduct a qualitative, instead of a quantitative, comparison.

## 5.2. Discussion

The three proposed techniques address symptoms of crosscutting functionalities, such as scattering and tangling, in quite different ways.

Overall, fan-in analysis and dynamic analysis show largely complementary results sets. This is an expected result, since the first technique focusses on identifying those methods that are called at multiple (scattered) places. However, when a method is called multiple times in a system, it is likely to occur in most (if not all) the execution traces, so that no specific use-case can be defined to isolate the associated functionality.

Identifier analysis is the least discriminating of the three techniques and has a large overlap with the other two techniques. When a concern can be identified through fan-in analysis and/or dynamic analysis, identifier analysis can often isolate it too, since a common lexicon is often used in the names of the involved methods. However, identifier analysis sometimes discards aspects reported by one of the other

two techniques due to the filtering that is applied to limit the number of concepts to be manually inspected. For example, aspects that are too small (in terms of number of involved methods) are often discarded by identifier analysis.

In conclusion, the results seem to indicate a big opportunity for the combination of the different methods, as will be discussed in Section 6.

## 5.3. Limitations

As a consequence of applying each technique to the same case, some of the limitations of the respective techniques have become obvious. For example, we obtained a better idea of potential 'false negatives', i.e. concerns that were not identified by a particular technique but that were identified by another. Below, we summarize some of these discovered limitations. In the next section we then discuss how these limitations may be overcome by combining the different techniques.

**Fan-In Analysis** mainly addresses crosscutting concerns that are largely scattered and have a significant impact on the modularity of the system. The downside of this characteristic is that concerns with a small code footprint and thus with low fan-in values associated, will be omitted. For example, the identification of *Observer* design pattern instances is dependent on the number of classes implementing the *observer* role. These classes contain calls to specific methods in the *subject* class for registering as listeners to the *subject*'s changes. The number of *observer* classes will determine to a large extent the number of calls to the registration method in the *subject* role. A collateral effect is the anticipated unsuitability of the technique for analyzing small case studies.

**Identifier Analysis** tends to produce a lot of detailed results. However, these results typically contain too much noise (false positives), so a more effective filtering of the discovered concepts, as well as of the elements inside those concepts, is needed. In addition, the discovered concepts are often incomplete, in the sense that they do not completely "cover" an aspect or crosscutting concern. Often, more than one concept is needed to describe a single concern, as was the case for the *Observer* aspect. The individual concepts themselves may also need to be completed with additional elements that are not contained in those concepts. This was the case for the *Undo* aspect: in addition to the methods with

‘undo’ or ‘undoable’ in their name, some of the methods calling these undo methods need to be considered as part of the core *aspect* as well.

**Dynamic Analysis** Among the known limitations of this technique, the two most important ones are that it is partial (i.e., not all methods involved in an aspect are retrieved), being based on a dynamic analysis, and it can determine only aspects that can be discriminated by different execution scenarios (e.g., aspects that are exercised in every program execution cannot be detected). Additionally, it does not deal with code that cannot be executed (e.g., code that is part of a larger framework, but is not used in a specific application).

## 6. Towards a combination of the techniques

In this section, we consider the ways in which the proposed techniques can be used to complement each other’s results. Our proposals for combination are based on our knowledge of the characteristics of the individual techniques, as well as on the experimental results presented above.

### 6.1. Combining the results

Since the three techniques are based on different characteristics, some aspects are found by one technique only (see Subsection 5.1). This does not imply that such aspects are less likely to be good aspects. For example, recall from Subsection 5.3 that dynamic analysis cannot deal very well with methods with a high fan-in and conversely fan-in analysis cannot deal very well with low fan-in methods. Therefore, it does make sense to take the union of all aspects discovered by these two approaches separately, thereby achieving a better coverage of the crosscutting concerns in the system. Examples of concerns detected by only one of these two techniques are *Contract enforcement / Consistent behavior* (fan-in) and *Bring to front / Send to back* (dynamic analysis).

Identifier analysis also detects concepts that are not detected by any of the other techniques. Given the large number of concepts identified by this particular technique, it is less clear if these results can simply be combined with the results of the other techniques. It remains to be investigated whether these detected concerns are false positives of identifier analysis, or actual concerns missed by the other techniques. In the former case, we could use the outcome of the other techniques to reduce these false positives reported by identifier analysis, by keeping an identifier analysis concept only when some of its elements are also reported by the other techniques as belonging to a concern.

### 6.2. Combining the techniques

Whereas the previous subsection explained how the end results of (some of) the techniques could be combined, this subsection focuses on how the techniques themselves could be combined in order to improve the quality of the results.

**Completion of aspects** As explained in Subsection 5.3, when identifying a particular crosscutting concern, fan-in analysis and dynamic analysis in fact only produce interesting seeds for that concern, which may serve as a starting point to discover the actual code addressing that concern. Developers need to browse the source code in order to “complete” the concern. Dynamic analysis in particular suffers from this problem as it only partial.

By combining the techniques in clever ways, this problem may be alleviated. For example, dynamic analysis may complete the output of fan-in analysis by other methods that label the same (dynamic analysis) concept as the ones calling that with the high fan in. In addition, the detailed information provided by identifier analysis may be used to complete both the results produced by dynamic analysis as well as fan-in analysis. For dynamic analysis, we may add methods belonging to the same (identifier analysis) concepts as those labeling the considered (dynamic analysis) concept. For fan-in analysis, we may add methods belonging to the same (identifier analysis) concepts as those with the high fan-in, or as those calling the method with the high fan-in.

As an example, let us consider the *Undo* concern. Since most of the involved methods are executed in all the considered scenarios, dynamic analysis can detect only a small subset of them. However, these can be associated with the (identifier analysis) concepts containing the recurring words (in our example, ‘undo’ and ‘undoable’) and the other methods in such concepts can be regarded as a completion of the candidate aspect detected by dynamic analysis.

**Reducing false negatives** Some of the generic concepts computed by dynamic analysis are discarded because they contain a high number of methods that can hardly be interpreted as belonging to one single concern. However, some commonality among them does exist, since they are shared by a subset of the execution traces. Thus, they potentially identify multiple, slightly different, crosscutting concerns. It would be possible to apply identifier analysis only to the methods reported by dynamic analysis in each such generic, large concept. Inspection of the resulting lattice could reveal the presence of clearly identifiable and now separated crosscutting functionalities.

**Grouping identifier analysis concepts** As can be seen from Table 2, more than one identifier analysis concept typically corresponds to one concern. The identifier analysis approach thus requires us to manually inspect the individual concepts to decide which of them are related to one and the same crosscutting concern. Information provided by the other techniques may help us to group concepts in a more automated way. For example, dynamic analysis produces a trace of different methods playing a role in a given concern. If these methods belong to different (identifier analysis) concepts, it may be a good idea to group these concepts. Consider the *Persistence* concern. The names of the methods

in the use-case specific concept reported by dynamic analysis include words that belong to different identifier analysis concepts (like, ‘file’, ‘storable’, ‘load’ or ‘register’). These could be unified into a unique candidate aspect.

### 6.3. Summary

We conclude that the three techniques can be combined in different ways. Since fan-in analysis and dynamic analysis are highly complementary, it makes sense to take the union of the discovered concepts. Both combined seem potentially very powerful for revealing the interesting seeds. Identifier analysis, on the other hand, seems like a good technique to complement and augment the discovered seeds with more information on where the discovered aspects are addressed in the code, in other words it may be used to extend the seeds into real concerns. Conversely, both fan-in analysis and dynamic analysis can be used to restrict the (often large) output produced by the identifier analysis.

## 7. Summary and future work

In this paper we took a closer look at three independently developed aspect mining techniques. Each of them has strengths and weaknesses. Fan in analysis is focused on those concerns that are implemented as scattered methods calls and manifest themselves as high fan in methods. Consequently, it fails to identify candidate aspects associated with low fan in. Identifier analysis can detect crosscutting whenever the involved methods share the same lexicon. However, it fails in presence of poor naming conventions or of concerns that share a common semantic context, rather than just a lexicon. Dynamic analysis relies on the possibility to isolate crosscutting functionalities through the execution of scenarios that exercise them. It fails whenever a functionality is present in all execution traces.

The properties used by the three techniques are orthogonal to each other. The experimental results obtained on a meaningful case study confirmed their complementary nature. This suggest the possibility of several useful combinations of these techniques. A simple combination strategy consists of taking the union. Yet, it is expected to be very effective and to actually increase the coverage, because of the different properties exploited by the three techniques. Since identifier analysis is based on the presence of meaningful words, that characterize the candidate aspect, it can be used to complete the other two techniques with all the methods sharing the same lexicon. Moreover, the presence of different words in fan-in analysis and dynamic analysis concerns might suggest the unification of identifier analysis concepts, that can be regarded as possibly associated to a single concern.

Our future work will be devoted to: (1) extending the comparison at the level of the seeds detected by each tech-

nique for the commonly identified concerns; (2) applying other aspect mining techniques to our case study; (3) implementing and evaluating the combination methods proposed in this paper; (4) defining an aspect mining evaluation framework, based upon our experience with the case study considered in this paper; (5) assessing the difficulty of moving from aspect identification to the actual refactoring towards aspects.

**Acknowledgements** Part of this collaboration was funded by RELEASE, a European Science Foundation scientific network.

## References

- [1] A. Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proc. of the First Int. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03)*, Nov. 2003.
- [2] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.
- [3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.
- [5] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.
- [6] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.
- [7] M. Marin, A. Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, The Netherlands, November 2004. IEEE Computer Society.
- [8] K. Mens and T. Tourwé. Delving source-code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 2005. To be published.
- [9] L. Moonen. Exploring software systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society Press, Sept. 2003.
- [10] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [11] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, 1999.
- [12] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.
- [13] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, The Netherlands, November 2004. IEEE Computer Society.

- [14] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, Chicago, Illinois, USA, September 2004. IEEE Computer Society.