Mining Aspectual Views using Formal Concept Analysis

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079
NL-1090 GB Amsterdam
The Netherlands
Email: tom.tourwe@cwi.nl

Kim Mens
Département d'Ingénierie Informatique
Université Catholique de Louvain
Place Sainte Barbe 2
1348 Louvain-la-Neuve, Belgium
Email: kim.mens@info.ucl.ac.be

Abstract

In this paper, we report upon an initial experiment using the technique of formal concept analysis for mining aspectual views from the source code. An aspectual view is a set of source code entities, such as class hierarchies, classes and methods, that are structurally related in some way, and often crosscut a particular application. Initially, we follow a lightweight approach, where we only consider the names of classes and methods. This simplistic technique already results in the discovery of interesting and meaningful aspectual views, leaving us confident that more complex approaches will perform even better, and should be studied in the future.

1 Introduction

The success of aspect-oriented programming [12] raises the important issue of how to turn existing software systems into aspect-oriented systems. Performing this transformation can be decomposed in two different tasks:

Aspect mining: identify the relevant aspects in the source code;

Aspect refactoring: define the appropriate aspects and restructure the base code in an aspect-oriented way.

Although each of these tasks could be performed manually, significant benefits would be gained from automating them. Preliminary support for aspect refactoring has only been proposed by [9]. Research in aspect mining is still in its infancy. Some advanced browsers have been proposed [8, 11], but these require a lot of pre-existing knowledge from the developers. To the best of our knowledge, *automated* support for aspect mining has not yet been studied. In this paper we propose a formally-founded approach as a first step in this direction..

More precisely, we report upon an initial experiment assessing the feasibility of *formal concept analysis* (FCA) [7] to discover *aspectual views* in the source code of an application automatically. We define an aspectual view as a set of source code entities that are structurally related in some way. These entities can be any source code artifact, such as a class hierarchy, a class, a method, a method parameter or an instance variable. The structural relation between these source code entities is arbitrary. An aspectual view can contain all source code entities that participate in a *Visitor* design pattern, for example. Hence, aspectual views offer a view on the source code that is often cross cutting and that complements the standard views offered by traditional development environments. As such, these views improve understandability and maintainability of the software.

The main goal of our experiment is to assess whether FCA could be used to analyse the source code and to discover meaningful aspectual views automatically. The contribution of this paper is thus twofold. First of all, we define a specific configuration of the FCA algorithm that can be used for mining. One of the main benefits of FCA is that it can be used for many different purposes. However, this means that the FCA algorithm should be tuned specifically for the purpose for which it is used. Second, we show which aspectual views can be discovered using this specific configuration. If the results are satisfactory, we might be able to extend our approach to perform real aspect mining, or at least use it as a technique to increase program comprehension [13].

The remainder of this paper is structured as follows. In the next section, we give a brief introduction to formal concept analysis. Section 3 introduces the approach we propose to discover aspectual views, while Section 4 discusses some of the more interesting views that were discovered. Section 5 discusses some limitations of the current approach, based on the results obtained. Section 6 presents an overview of related work. Section 7 discusses some extensions to our approach, and we draw our conclusions in

Entity	smalltalk	call	term	visit	make	underscore	variable	constant
callTermVisit:	-	√	√	√	-	-	-	-
smalltalkTermVisit:	√	-	√	√	-	-	-	-
constantVisit:	-	-	-	√	-	-	-	√
makeUnderscoreVariable	-	-	-	-	√	√	√	-
UnderscoreVariable	-	-	-	-	-	√	√	-

Table 1. Class and method names and their string properties

Section 8.

2 Formal Concept Analysis

The technique of formal concept analysis is fairly simple. Starting from a (potentially large) set of elements and properties of those elements, FCA determines maximal groups of elements and properties. These maximal groups are called concepts. Every such concept consists of a set of elements that have one or more properties in common and such that no other elements have those properties nor are there any other declared properties they have in common.

As an example, consider as elements the methods `callTermVisit:`, `smalltalkTermVisit:` `constantVisit:` and `makeUnderscoreVariable` and the class `UnderscoreVariable`, and as properties of these elements their string components, such as `call`, `smalltalk`, `term`, `visit` and so on (See Table 1). The concept lattice, that is computed by the FCA algorithm based on these elements and properties, is depicted in Figure 1. The bottom concept contains those elements that share all properties. Since there is no such element in our example, that concept contains no elements. Similarly, the top concept contains the properties shared by all elements. Since there is no such property, the concept contains no properties. But the concept ({ constantVisit:, smalltalkTermVisit:, callTermVisit: }, { visit }), for example, groups all visitor methods. Similarly, the concept ({ smalltalkTermVisit:, callTermVisit: }, { term, visit }) groups all visitor methods for terms.

3 Experiment

Now that we have explained the essence of FCA, we turn our attention to how it could be used to mine source code for aspectual views and give a detailed explanation of the setup of our initial experiment.

3.1 Approach

We divided our aspectual view mining process into the following different phases:

Generate elements and properties. In this first phase, we inspect the source code and generate the elements and properties to be considered, in a format that is processable by the FCA algorithm. The elements can be any source code artifact, such as classes, methods, variables, individual statements, etc. In our initial experiment, we only consider classes, methods and formal parameters, and their associated properties are the substrings appearing in their name (as will be explained later on). In general, however, we could consider any structural property of these elements, derived from the source code. Such properties could be the class in which a method is defined, or the other methods a particular method invokes.

Generate the concept lattice. In the second phase, we apply a concept analysis algorithm to the elements and properties generated by the previous phase, resulting in a concept lattice.

Filter out unimportant concepts. If a software system of significant size is considered, the number of concepts generated by the FCA algorithm (i.e., the number of concepts in the concept lattice) may be quite high. Therefore, the lattice needs to be inspected and analysed further in order to find the information of interest. We define a number of heuristics that enable us to discard unimportant concepts automatically. These heuristics can depend on the particular elements and properties that are considered, but can they can also be more general. For example, if the top and bottom concept contain no elements or properties, respectively, they can be filtered out. Moreover, we are not interested in concepts containing only one element, since such concepts represent one single source code artifact that has no relations to other artifacts (according to the selected properties).

In our initial experiment, these general filters are the only ones we used. As a result, the number of remaining concepts is still quite high, so there is still a need to define more (domain-)specific filters. Given the preliminary status of our research, we chose to explore all remaining concepts manually, to enable us to gradually define more specific filters.

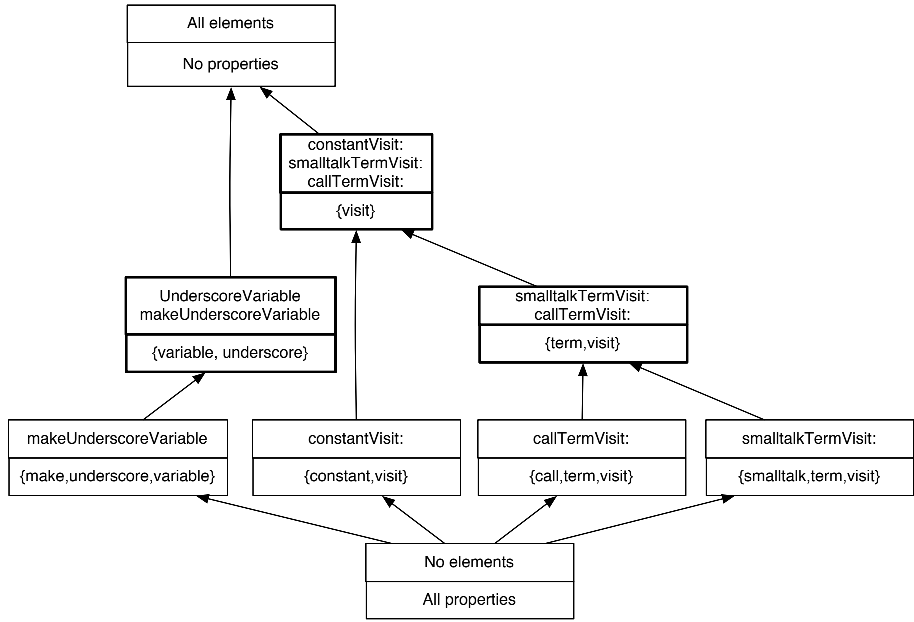

Figure 1. Concept lattice generated based on Table 1

Analyse and classify remaining concepts. The concepts that remain after filtering are analysed and classified, according to some desired criteria. When dealing with concepts containing only methods, for example, we can classify them according to the classes in which they are defined. In this way, we can distinguish between concepts containing methods that are all defined in the same class, concepts containing methods all defined in a particular hierarchy, or containing cross-cutting methods that are defined in different classes not related by inheritance. Of course, other criteria can be defined, since these can depend upon the particular situation at hand.

Display concepts Finally, once the concepts have been analysed and classified, they are ready to be inspected by the developer(s). To this extent, we display the concepts in a graphical user interface, that allows a developer to browse, inspect and even regroup the concepts, as desired.

3.2 Setup of the Experiment

In our initial experiment, we deliberately chose a rather lightweight technique to discover related classes and methods in the source code of a software system. The software system we studied was Soul [19], a logic programming environment written in VisualWorks Smalltalk. We chose this application, because it is a small to medium sized application (consisting of 111 classes and 1335 different method definitions), and because we know its implementation quite well. This allows us to fine tune the FCA algorithm as much as needed to achieve satisfactory results. Note that this does not mean that the algorithm will only work for this particular application. Once a well-working configuration of the algorithm has been identified, it can of course be reused for mining other applications.

The elements we feed into the FCA algorithm are all classes the Soul system implements, as well as all methods these classes define. The properties we consider for those elements are based on a decomposition of their names in relevant substrings:

- For a class, we consider its name and, according to the capitals occurring in it, split it into substrings. For example, the properties associated with the class `QuotedCodeConstant` are that its name has the substrings 'quoted', 'code' and 'constant'.
- For a (Smalltalk) method, we also consider as properties that it contains certain substrings, and generate these properties by taking the method's name and splitting it according to its keywords and the capitals occurring in them. For example, the method `unifyWithDelayedVariable:inEnv:myIndex:hisIndex:inSource:` has 5 keywords, which are split into the following substrings: 'unify', 'delayed', 'variable', 'env', 'index' and 'source'. Note that we discard strings with little conceptual meaning, such as 'with' and 'in'. In addition to a method's name, we also take into account (the names of) its formal parameters. For

example, the `inEnv:` keyword of the above method defines a formal parameter `anEnvironment`, for which we obtain the substring 'environment'. Of course, if the name of the formal parameter contains multiple capitals, multiple substrings and their corresponding properties will be generated.

The motivation behind this rather simplistic scheme for generating properties for classes and methods is that we hope that standard Smalltalk coding conventions can be relied upon to group related classes and methods into aspectual views. Particular coding conventions we are aiming at are, amongst others, the following:

- Method overriding, or polymorphism in general, is an intrinsic property of any object-oriented programming language, that is based on a simple naming convention. That is, two classes can only be used polymorphically, if (some of) their methods have exactly the same name. By exploiting this convention, we can detect classes that can be used polymorphically. Note that these classes need not necessarily reside in the same class hierarchy. Since Smalltalk is dynamically typed, it allows any class to be substituted for another, as long as it defines the required methods.
- Since Smalltalk is a dynamically typed language, formal parameter definitions do not have an associated (static) type. To improve program understanding, a widely accepted programming convention is to name a parameter after the class whose objects should be passed. For example, the keyword `inEnv:` discussed above should be passed an instance of the `Environment` class. To reflect this, the developer named the parameter 'anEnvironment'. Relying on this convention, we hope to be able to relate a class to the methods that use its objects.
- Some programming idioms, such as the *double dispatch* idiom, and design patterns like *Visitor* or *Abstract Factory*, also make heavy use of so-called 'intention revealing selectors', meaning that the names of the methods they define are chosen meticulously to reflect the intention of the pattern. In particular, many of these methods include the name of the class they are dealing with, such as `makeTermSequence`, which instantiates a new `TermSequence` object, or `unifyWithDelayedVariable: inEnv: myIndex: hisIndex: inSource:`, which implements unification for the `DelayedVariable` class. Relying on such conventions, we hope to identify patterns such as these. Also, it may allow us to relate a class to the methods that deal with it (without having to do any type inferencing).

After filtering, the remaining generated concepts were classified automatically according to the following criteria:

class name in keyword concepts contain both classes and methods, where the class names form a substring of the methods' names.

class name in parameter concepts contain both classes and methods, where the class names occur in one or more parameter names defined by the methods.

classes only concepts contain only classes.

accessor concepts contain only methods, defined in the same class, that are named after an instance variable of that class.

methods in single class concepts contain only methods that are all defined in the same class, but that are not accessor methods.

hierarchy method concepts contain only methods, defined in different classes, where all these classes share a common superclass not equal to `Object`[1], unless Object itself also defines the method.

crosscutting method concepts contain only methods, defined in different classes, where the common superclass of all these classes is `Object` and where none of methods in the concept is defined on the `Object` class itself[2].

In what follows, we will present some of the concepts that we identified as important, each time indicating to which category they belong.

4 Discovered Aspectual Views

In this section, we discuss some of the most interesting aspectual views that were discovered by applying the above approach on the Soul system. Subsequent subsections discuss aspectual views dealing with *programming idioms, design patterns, features* and *code duplication*.

The concept lattice was computed using 1446 different elements and 516 properties in total. The fact that the number of properties is one third of the number of elements already indicates that many properties are shared among elements. The FCA algorithm created 1212 concepts in total, of which 760 remained after filtering.

4.1 Programming Idioms

As a good indication that our approach seems to produce some relevant results, we discovered several occurrences of typical programming idioms, of which we discuss the *accessor methods* and *polymorphic methods* programming idioms in this section.

[1] the superclass of all classes in the system

[2] which would be a degenerate case of a *hierarchy method* concept

4.1.1 Accessor Methods (*accessor concepts*)

Accessor methods are methods defined by a class to manipulate its instance variables. For each instance variable of the class two methods are defined: an *accessing* method, that simply returns the value of the particular variable, and a *mutator* method that can change this value. In Smalltalk it is custom to use the name of the instance variable for both the accessor and mutator method.

For example, the following concept we discovered groups the `callStack` accessing and `callStack:` mutator methods of the `callStack` instance variable defined in the `Environment` class:

```
Environment >> callStack
    ^ callStack
Environment >> callStack: aStack
     callStack := aStack
```

The structural form of these methods is typical for accessor methods. However, several other forms exist, and were identified by the FCA algorithm as well. For instance, the following concept was identified and groups accessors for the `outputStream` instance variable of the `LogicTestNotifier` class. The difference with the former accessors is that this instance variable is initialised in a lazy way (i.e. if it is accessed before it is initialised, it is first initialised and only then its value is returned):

```
LogicTestNotifier >> outputStream
    outputStream isNil
           ifTrue: [outputStream := Transcript].
    ^outputStream
LogicTestNotifier >> outputStream: aStream
    outputStream := aStream
```

4.1.2 Polymorphism (*hierarchy method concepts*)

Because the properties we considered were based on substrings, methods with the exact same name are of course grouped by the FCA algorithm. Such methods allow the classes that define them to be used polymorphically. Not surprisingly, a number of such concepts were identified, since any well-designed object-oriented software application uses method overriding and polymorphism, and Soul is not an exception. Such concepts are interesting, because they offer a view on methods that implement related behaviour, not offered in traditional IDEs,

It is interesting to note that two different categories of concepts containing polymorphic methods can be identified: concepts that group methods defined in the same class hierarchy and concepts that group methods not defined in the same class hierarchy. A standard example of the former category is the concept containing all methods of class `AbstractTerm` that are overridden in its various subclasses: the selector `smalltalkBlockString` that is implemented in `AbstractTerm`, `Variable`, `SpliceTerm` and `QuotedCodeConstant`, or the selector `transitiveLookupIn:startAt:`, implemented in 14 classes of the `AbstractTerm` hierarchy, for example. An example of the second category is the concept containing the `prettyListPrintOn:scope:` method, that is defined in the `ListTerm`, `UnderscoreVariable` and `EmptyListConstant` classes. The common superclass of these classes is `AbstractTerm`, which does not define that method. This situation can be due to different reasons: either its is a case of a bad interface design or of bad naming of some methods, or it is a case of behaviour that is crosscutting a class hierarchy. Here, the former case occurs.

4.2 Design Patterns

This section illustrates some design pattern occurrences that were identified by the FCA algorithm. It should be noted that Soul contains only three design pattern instances, two of which were discovered. The one that was not discovered was an instance of the *Builder* pattern, used by the Soul parser. The reason why this instance was not discovered will be explained in Section 5.

4.2.1 Visitor Design Pattern (*hierarchy method concepts*)

Soul uses an instance of the *Visitor* design pattern to perform a number of operations on logic terms. These terms are represented as subclasses of the `AbstractTerm` hierarchy, while the visitor hierarchy is defined by the `SimpleTermVisitor` class.

The presence of the Visitor design pattern was recognised by two classifications containing concepts that were detected separately.

The first classification concerns the visitor hierarchy. It groups all methods that are defined in the `SimpleTermVisitor` hierarchy and that implement behaviour to be executed when a particular term is visited (e.g. these are the *visit method* participants in the Visitor design pattern). The classification consists of a number of concepts, each of which contains all methods defined by subclasses of class `SimpleTermVisitor`, dealing with one particular term. For example, as shown in Figure 2, one such concept is defined by the properties 'visit' and 'compound', and contains the various implementations of the `compoundVisit:` method, defined in the `SimpleTermVisitor` hierarchy and responsible for implementing behaviour associated to a `CompoundTerm` object. In this particular case, the concept consists of four `compoundVisit:` methods, implemented in the classes `SimpleTermVisitor`, `LexicalAddressVisitor`, `CopyingVisitor` and `CompoundTermRenamingVisitor`.

The second classification concerns the *element* hierarchy of the visitor design pattern, and contains only one concept.

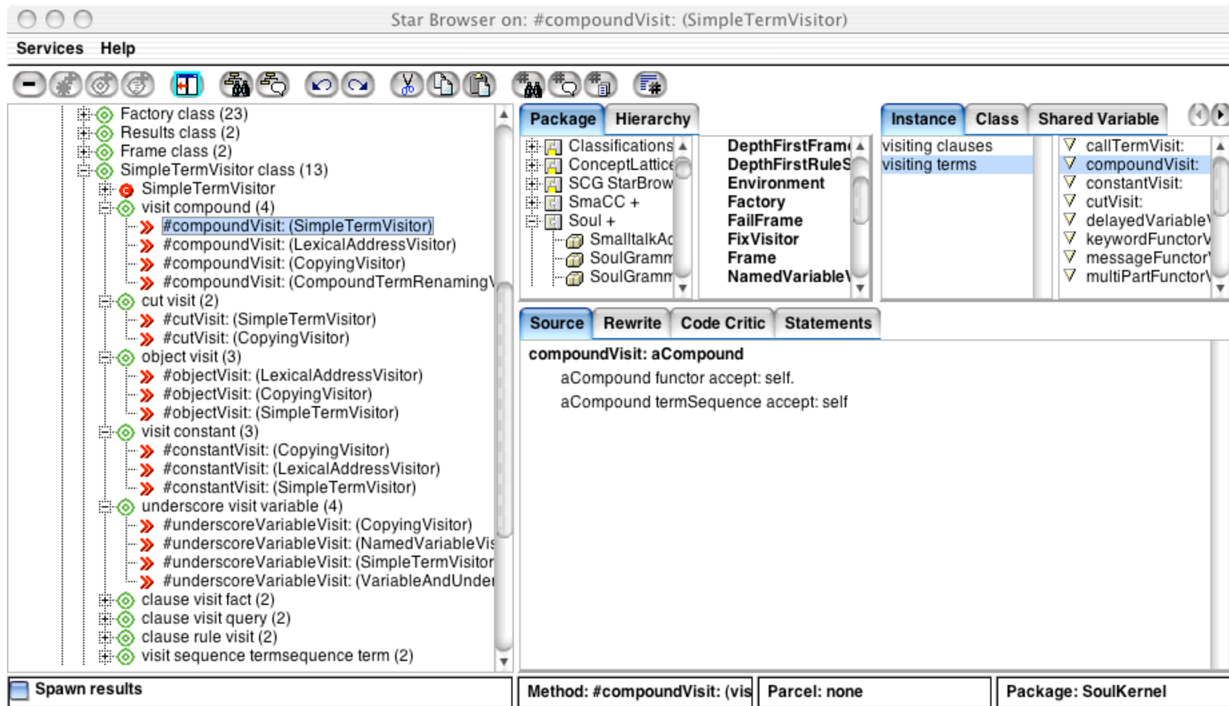

Figure 2. Classification of concepts related to the Visitor design pattern

This concept groups all `accept` methods that are defined by subclasses of the `AbstractTerm` class. These methods are responsible for calling the appropriate method, corresponding to the term being visited, in the supplied visitor object. They are grouped by the FCA algorithm based on the 'visitor' and 'accept' substring properties. This concept thus forms an illustration of a concept that takes into account both the method's name and the name of the formal parameter it defines, since the `accept:` method in all classes defined a formal parameter `aVisitor`.

4.2.2 Abstract Factory Design Pattern (*hierarchy method concepts*)

Soul also uses an instance of the *Abstract Factory* design pattern, that is responsible for creating new instances of many different classes in the system, among others, subclasses defined in the `AbstractTerm` and `HornClause` hierarchies. The instance defines an interface for creating such objects in the `Factory` abstract superclass, and implements this interface concretely in the `StandardFactory` subclass. The interface consists of methods such as `makeCut` and `makeQuotedCodeTerm`, that are responsible for instantiating `Cut` and `QuotedCodeTerm` objects.

The presence of this abstract factory design pattern instance was recognised by one classification that groups different concepts, and that looks similar to the first classification for the visitor design pattern. The classification groups all concepts that contain methods that instantiate new objects. Each such concept groups an abstract method of the `Factory` class and its concrete counterpart defined in the `StandardFactory` class. For example, a concept based on the properties 'make' and 'cut' is identified, that contains the two implementations of the `makeCut` method in the `Factory` hierarchy.

4.3 Features

In addition to occurrences of well-known programming idioms and design patterns, a number of interesting *features* were also identified. These features manifest themselves as concepts that group related source code entities that are spread among many different classes.

4.3.1 Unification (*hierarchy method concepts*)

Since Soul is a logic programming environment, it implements a general logic unification algorithm. This algorithm is implemented in the `AbstractTerm` hierarchy, and is responsible for unifying the different kinds of terms. The algorithm is spread among different subclasses, each of which implements a `unifyWith: inEnv: myIndex: hisIndex: inSource:` method, that tries to unify the receiver

with the term passed to the `unifyWith:` keyword. To this extent, it uses the double dispatch mechanism. The `unifyWith: inEnv: myIndex: hisIndex: inSource:` method defined in the `UnaryMessageFunctor` class, for example, thus simply sends a `unifyWithUnaryMessageFunctor: inEnv: myIndex: hisIndex: inSource:` message to the term passed in the first argument.

One concept is identified that groups all implementations of the `unifyWith:inEnv:myIndex:hisIndex:inSource:` methods spread among the `AbstractTerm` hierarchy. Several other concepts are discovered grouping the different parts of the unification algorithm for particular subclasses. Typically, these concepts consists of three method definitions: one method defined in the `AbstractTerm` class, one in the `Variable` class and one in the specific term class. For example, a concept exists that groups the `unifyWithQuotedCodeTerm: inEnv:myIndex: hisIndex: inSource:` methods implemented in the `AbstractTerm`, `QuotedCodeTerm` and `Variable` classes, as shown in Figure 3.

From an aspect-related point of view, these concepts are particularly interesting. Clearly, different parts of the unification algorithm (e.g. the way `QuotedCodeTerm`s or `CompoundTerm`s should be unified) are spread over different classes, and crosscut the existing decomposition. The aspectual views allow us to regroup all methods that implement the algorithm into one single place. Consequently, this improves program understanding and maintenance: when changes are needed to parts of the algorithm, the view can be used to verify what needs to be changed, how this affects the algorithm and what the possible impact may be.

4.3.2 Crosscutting Class-Related Behaviour (*class name in keyword and class name in parameter*)

Many concepts are identified that group a class and all methods that are related to that specific class. The class and the methods are grouped because the name of the class is a substring of the name of the method, or the name of the class appears in one of the formal parameters of the method. For example, a concept exists that is based on the 'delayed variable' substring property, and that contains the `DelayedVariable` class and the following methods:

- `delayedVariableVisit:` that forms part of the Visitor design pattern;
- `buildDelayedVariable` that forms part of the Builder design pattern;
- `makeDelayedVariable` that forms part of the Abstract Factory design pattern;
- `unifyWithDelayedVariable: inEnv: myIndex: hisIndex: inSource:` that forms part of the unification algorithm.

Typically, such concepts always contain these four methods, albeit with a different class. Other methods may be present as well, of course. For example, some concepts contain a class testing method, such as the concept containing the `Variable` class, which contains an `isVariable` method, besides `variableVisit:`, `buildVariable`, `makeVariable` and `unifyWithVariable:inEnv:myIndex:hisIndex:inSource:` methods. Other classes also implement a similar testing method.

Sometimes, not all of the methods in the above enumeration are present in the concept. Most of the time, this is a sign that common behaviour in the class hierarchy has been factored out. For example, the concept containing the `QuotedCodeConstant` class does not contain the methods belonging to the Visitor design pattern and the unification algorithm. This is due to the fact that the `Constant` superclass of the `QuotedCodeConstant` class deals with these issues.

Like the concepts in the previous subsection, the concepts identified here contain behaviour that crosscuts the existing decomposition. They contain classes and methods related to these classes, that are implemented in totally different classes. Whenever we change the name of a class, its instantiation interface or its unification scheme, we can use this aspectual view to assess the impact of the change on these methods, for example. Also, the view helps us to understand the structure and part of the inner working of the application: if we add a new class, we know which methods we should implement.

4.4 Code Duplication (*methods in single class concepts and crosscutting method concepts*)

An interesting phenomenon we observed is that we can detect some instances of copy and paste reuse. Several of the concepts that were discovered contains methods with nearly the same name, that also define nearly the same behaviour. This seems logical: methods that do the same thing should have the same name. In some cases, duplicated code may just point out a bad design: methods containing duplication are defined in the same class, which makes it easy to refactor the code and remove the duplication. In other cases, the duplicated code is spread over different classes, and is not as easy to factor out. These cases may indicate real crosscutting code that can only be factored using aspects.

An example of a concept of the former category is the following: two methods of the `Variable` class were grouped because they have a similar name, and they clearly have nearly the same implementation:

```
unifyWithMessageFunctor: anMPFunctor inEnv: anEnv
myIndex: myIndex hisIndex: hisIndex inSource: inSource

    | val |
    self halt: 'todo!'.
    val := anEnv lookup: self.
    val isNil
```

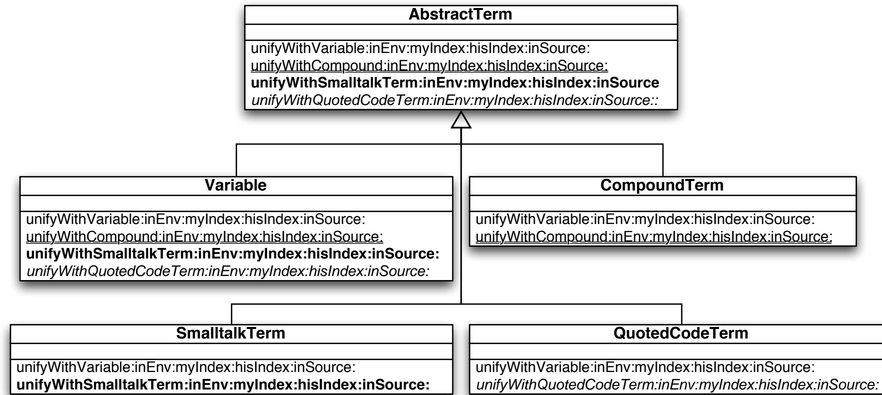

Figure 3. Unification in the Soul implementation

```
        ifTrue:
            [anEnv add:
              (Factory current
                     makeBinding var: self
                                 value: anMPFunctor).
             ^ true]
        ifFalse:
            [^ val unifyWithMessageFunctor: anMPFunctor
                   inEnv: anEnv myIndex: val envIndex
                   hisIndex: hisIndex
                   inSource: inSource ]

unifyWithKeywordFunctor: anMPFunctor inEnv: anEnv
myIndex: myIndex hisIndex: hisIndex inSource: inSource

    | val |
    self halt: 'todo!'.
    val := anEnv lookup: self.
    val isNil
        ifTrue:
            [anEnv add:
                (Factory current
                     makeBinding var: self
                                 value: anMPFunctor).
             ^ true]
        ifFalse:
            [^ val unifyWithKeywordFunctor: anMPFunctor
                   inEnv: anEnv]
```

The fact that there is a halt statement inside the methods may point out that this code is still under active development.

An example of crosscutting code duplication can be found in the implementation of the `recursiveDefinitionRepository` method, defined by the `MultiPartFunctor` and `TermSequence` classes, among others. These methods contain exactly the same code, that can not be factored out in the common superclass `AbstractTerm`, because that class provides another implementation for that method.

5 Limitations

Because we use a lightweight technique that only considers substrings of class and method names, it should come as no surprise that our approach suffers from a lot of false positives. Even despite the fact that we tried to filter out unimportant concepts by using some heuristics. A large number of the 760 identified concepts are not really meaningful aspectual views, and are merely generated because some particular substring is shared between the view's elements. For example, a concept containing the classes `ObjectWrapper` and `ClauseWrapper` exists, simply because those two classes share the substring 'wrapper' in their name.

Due to a similar reason, false negatives occur as well. Some source code artifacts that actually belong together are divided over different concepts, simply because they do not share the exact same substring in their name. For example, a new language feature was incorporated in the latest Soul version, that deals with language symbiosis between Smalltalk and Soul. Classes that implement this feature sometimes contain the substring 'symbiotic' (such as the class `SymbioticMessageTerm`), and sometimes contain the substring 'symbiosis' (such as the class `SymbiosisFactory`). These classes will thus not end up in the same concept, although they belong together conceptually.

Both problems can be alleviated up to some extent by using some kind of a *domain ontology*. This ontology should contain information about the domain in which the FCA algorithm is applied, to filter out unimportant properties (such as the 'wrapper' substring), and linking related properties (such as the 'symbiotic' and 'symbiosis' substrings) so that they end up in the same concept.

False negatives also occur because our technique relies heavily on coding conventions and programming idioms. If these are not strictly adhered to, some important concepts can be missed. As an example, the Soul parser makes use of the Builder design pattern for building parse trees, but this pattern was not detected by our tool. This is due to the fact that there is only one participant in that particular pattern instance, the `SoulParseTreeBuilder` class, that de-

fines the default behaviour and is never subclassed. Consequently, there is no class hierarchy that contains related behaviour, nor are there polymorphic methods in that class. Several concepts are generated for that class, but each one contains only that class or only one of its methods. As such, these concepts are filtered out.

The most obvious limitation of our current approach is that we do not mine for real aspects, only for aspectual views. Although these views group related entities that are often spread throughout the code, and are hence crosscutting, they are most of the time not aspects in the real sense of the word. E.g. there is often no non-functional requirement implemented by the methods contained in an aspectual view that should better be captured in an aspect. It has been argued by some [10], however, that design patterns such as the Visitor and the Abstract Factory can better be captured by an aspect. Nonetheless, the obtained results are quite interesting from a program understanding point of view. In Section 7, we will elaborate on a more sophisticated approach that we want to consider to mine for real aspects.

6 Related Work

The use of formal concept analysis in software engineering is of course not new. An overview of the use of this technique for several purposes can be found in [16]. We only list the ones most related to our work here. [15] used concept analysis to re-engineer class hierarchies in C++. [17] used concept analysis to detect instances of design patterns in the source code, while [2] investigates the source code by means of concept analysis. to study how classes in object-oriented programs are reused. [5] used formal concept analysis to reveal the structure of classes.

In the context of migrating traditional applications to object-oriented applications, several authors have used concept analysis to identify meaningful objects [18, 4, 14]. Our technique can be considered an extension of such techniques, since one of our goals is to identify aspects as well, which are an extension of objects. [6] presents an approach for identifying source code entities specific for particular *features*, or functional requirements, of an application. Since all entities implementing a feature can be scattered over the source code, the implementation of such a feature can be considered an aspect. Consequently, the technique can also be suited as an aspect-mining technique, although this still has to be investigated.

Several other authors have used strings and identifiers as a lightweight means to reverse engineer source code. [3] analyses function identifiers by considering their lexical, syntactical and semantical structure. Their goal is to provide a meaningful summary of a function's behaviour and to group functions according to domain concepts. Although this approach and our approach are similar to a large extent, their goals clear differ from our goals. [1] analyses file names in order to identify particular subsystems of a legacy application. This approach is entirely different from our approach, since we consider method and class names, and we are less interested in recovering information at the architectural level.

7 Future work

Given that our initial experiment was based on a lightweight approach, but still allowed us to obtain some interesting and promising results, the question is raised how we should continue. Most importantly: how can we modify and refine our approach to be able to detect real aspects, and not only aspectual views?

An obvious improvement to our approach would be to go beyond considering only substrings of class and method names. More structural information can and should be taken into account. We are currently investigating how parse tree information can be used, for example. We consider individual parse tree statements as the properties of methods, which enables the FCA algorithm to group methods that share some statements. In this way, duplicated code can be detected more easily, which may point out the need for refactoring or aspects. For example, when only the first and the last line of code is duplicated in a number of methods, this may be an ideal candidate for an *around* advice in AspectJ.

The approach we advocated in this paper consists of identifying potentially interesting elements and properties, and then verifying whether aspects or aspectual views were discovered. A complementary approach can be interesting as well: taking an existing application, built using aspect-oriented technology, and deduce which elements and properties the FCA algorithm would need in order to detect the aspects as concepts. This would also allow us to fine tune and tweak the approach to ensure that false negatives and false positives are avoided as much as possible.

Although we experimented with a Smalltalk application only and we relied on specific coding conventions, our technique is programming language independent. Similar conventions then the ones we relied on also exist in Java. For example, the names of accessor methods should start with a particular prefix (e.g. `get` or `set`), and the name of visit method participants in the Visitor design pattern should also contain the class to be visited as a substring. Given the widespread use of AspectJ, and the sheer number of applications using aspect technology, we would thus also like to experiment with Java code.

8 Conclusion

In this paper, we proposed an approach to aspectual view mining using the technique of formal concept analysis. The experiment we conducted to validate this approach was based on a lightweight technique considering only class and method names. Although we have no full proof that concept analysis is able to discover real aspects in the source code, we do think that the results are quite promising: we were able to discover some programming idioms and design pattern instances in the code, as well as some interesting cross-cutting features. From a program understanding and maintenance point of view, this is already quite interesting. Moreover, because these results were obtained despite the fact that the properties we considered were quite simplistic, we are confident the approach can be tailored for real aspect mining as well. Significantly better results might be obtained when considering more complex properties, potentially at the cost of more computational time.

References

[1] N. Anquetil and T. C. Lethbridge. Recovering Software Architecture from the Names of Source Files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.

[2] G. Arévalo and T. Mens. Analysing object oriented framework reuse using concept analysis. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, 2002.

[3] B. Caprile and P. Tonella. Nomen Est Omen: Analyzing the Language of Function Identifiers. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1999.

[4] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying Objects in Legacy Systems using Design Metrics. *Journal Of Systems and Software*, 44(3):199–211, 1999.

[5] U. Dekel and Y. Gil. Revealing Class Structure with Concept Lattices. In *Proc. 10th Working Conference on Reverse Engineering*, 2003.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *Transactions on Software Engineering*, 29(3):210–224, 2003.

[7] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[8] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, University of California, Department of Computer Science and Engineering, 3, 2000.

[9] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35. Springer Verlag, 2003.

[10] J. Hannemann and G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. ACM Press, 2002.

[11] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 178–187. ACM Press, 2003.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] K. Mens and T. Tourwé. Conceptual Code Mining – Mining for Source-Code Regularities with Formal Concept Analysis. In *Submitted to the European Smalltalk User Group Conference (ESUG)*, 2004.

[14] P. Newcomb and G. Kotik. Reengineering Procedural into Object-Oriented Systems. In *Proceedings of the 2nd Working Conference On Reverse Engineering (WCRE)*. IEEE Computer Society, 1995.

[15] G. Snelting and F. Tip. Reengineering Class Hierarchies Using Concept Analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1998.

[16] T. Tilley, R. Cole, P. Becker, and P. Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *Proc. 1st International Conference on Formal Concept Analysis*, 2003.

[17] P. Tonella and G. Antoniol. Inference of object oriented design patterns. *Journal of Software Maintenance - Research and Practice*, 13(5):309 – 330, September - October 2001.

[18] A. van Deursen and T. Kuipers. Identifying Objects using Cluster and Concept Analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. ACM Press, 1999.

[19] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.