Delving Source-code with Formal Concept Analysis

Kim Mens *

Département d'Ingénierie Informatique (INGI)
Université catholique de Louvain (UCL)
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Tom Tourwé

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract

Getting an initial understanding of the structure of a software system, whether it is for software maintenance, evolution or reengineering purposes, is a nontrivial task. We propose a lightweight approach to delve a system's source code automatically and efficiently for relevant concepts of interest: what concerns are addressed in the code, what patterns, coding idioms and conventions have been adopted, and where and how are they implemented. We use formal concept analysis to do the actual source-code mining, and then filter, classify and combine the results to present them in a format that is more convenient to a software engineer. We applied a prototype tool that implements this approach to several small to medium-sized Smalltalk applications. For each of these, the tool uncovered several design pattern instances, coding and naming conventions, refactoring opportunities and important domain concepts. Although the tool and approach can still be improved in many ways, the tool does already provides useful results when trying to get an initial understanding of a system. The obtained results also illustrate the relevance and feasibility of using formal concept analysis as an efficient technique for source code mining.

Key words: Source-code mining, formal concept analysis, software classification.

* Corresponding author.
Email addresses: `Kim.Mens@info.ucl.ac.be` (Kim Mens), `Tom.Tourwe@cwi.nl` (Tom Tourwé).
URLs: `http://www.info.ucl.ac.be/people/cvmens.html` (Kim Mens), `http://www.cwi.nl/~tourwe` (Tom Tourwé).

Preprint to appear in Elsevier Computer Languages, Systems & Structures 29, 2005

1 Introduction

When maintaining or evolving a software system it is important to gain some knowledge of its overall design first. Demeyer et al. [1, Chapter 4] explain how obtaining such an *initial understanding* is crucial for the success of a reengineering project, and discuss some techniques and lightweight source-code analysis tools to get such an understanding. The tool proposed in this paper can be seen as another such tool in the software engineer's toolbox.

The tool implements a bottom-up approach that can help in getting an initial idea of the coding conventions, idioms and patterns used in the source code of a software system. In a sense, it is related to the "Study the Exceptional Entities" reengineering pattern documented in [1, Chapter 4], but in a complementary way, as it focusses on finding commonalities in the structure of the source code, rather than potential design problems.

Our tool builds on the technique of *formal concept analysis* [2], which has many known applications in data analysis and knowledge processing, and some in software engineering. The essence of our contribution lies not in the idea of applying formal concept analysis (FCA) to source code, but in our particular choice of elements and properties for the FCA algorithm, and how we filtered and classified the discovered concepts in order to mine a system's source code in a lightweight way that is independent of the actual system being analyzed.

Although the proposed approach can still be improved in many ways, and in spite of its apparent simplicity, our case studies show that it allows us to delve Smalltalk source code for many interesting symptoms of good design, like design patterns, programming idioms and naming and coding conventions. It also allows us to discover symptoms of bad design which may provide opportunities for refactoring, as well as some features of which the implementation is spread throughout the source code. Most of the discovered information provides a good starting point for understanding the source code in more detail.

The remainder of this paper is structured as follows. Section 2 briefly introduces the mathematical technique of formal concept analysis. In Section 3 we explain our approach and how we use formal concept analysis to delve the source code for relevant concepts of interest. Section 4 briefly presents the tool and Section 5 gives an overview of the experiments we conducted on five different small to medium-sized case studies and presents the design symptoms we discovered. Sections 6 and 7 discuss related and future work. We conclude the paper in Section 8.

2 Formal Concept Analysis

Formal concept analysis (FCA) [2] is a branch of lattice theory that can be used to identify meaningful groupings of *elements* that have common *properties.*[1] The FCA algorithm takes as input a relation, or Boolean table, T between a (potentially large, but finite) set of elements and a set of properties of those elements. An example of such a table is given in Table 1, in which different programming languages and properties are related. A mark in a table cell means that the programming language in the corresponding row has the property of the corresponding column.

Table 1
Programming languages and their supported programming paradigms.

Progr. language	OO	Functional	Logic	Static typing	Dynamic typing
Java	√	-	-	√	-
Smalltalk	√	-	-	-	√
C++	√	-	-	√	-
Scheme	-	√	-	-	√
Prolog	-	-	√	-	√

Taking such a table T as input, the FCA algorithm 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, and
- no other property outside the group holds for all elements in the group.

Intuitively, a *concept* corresponds to a maximal 'rectangle' in the table T, up to a permutation of the table's rows and columns.

All concepts are ordered into a *concept lattice*, an example of which is depicted in Figure 1. The lattice's bottom concept contains those elements that have all properties. Since there is no such programming language in our example, that concept contains no elements. Similarly, the top concept contains those properties that hold for all elements. Again, there is no such property. Other concepts represent related groups of programming languages, such as the concept *({ Java, C++ }, { static typing, object oriented })*, which groups all statically-typed object-oriented languages.

[1] As in Arèvalo et al. [3], in this paper we prefer to use the terms *element* and *property* instead of *object* and *attribute* used in traditional FCA literature, because these latter terms already have a very specific meaning in OO software development.

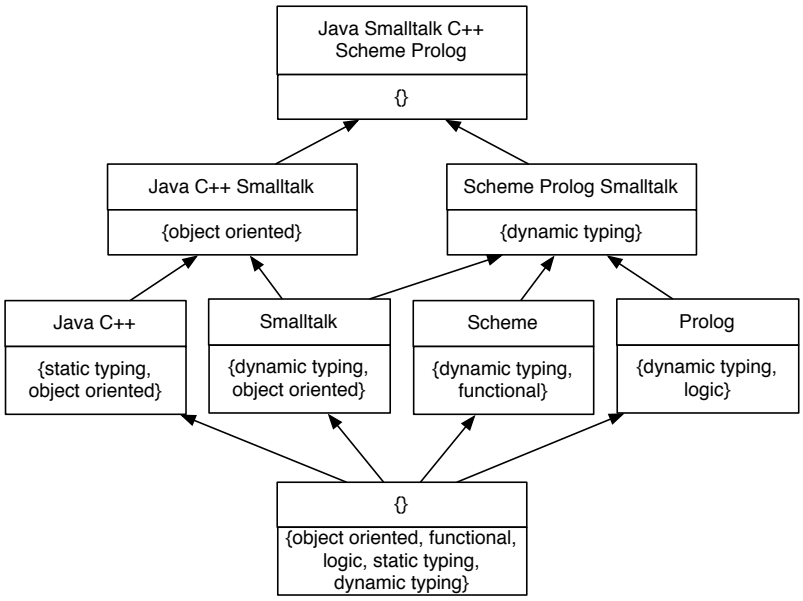

Fig. 1. Concept lattice corresponding to Table 1.

For more details on formal concept analysis we refer to [2]. The next section explains the details of our approach to use FCA for source code mining.

3 Delving Source Code

When applying FCA for delving Smalltalk source code, we first have to choose the elements and properties to compute the concept lattice (§3.1). When computing the lattice (§3.2), lots of concepts are produced, many of which are irrelevant or redundant. Therefore, we filter the discovered concepts (§3.3) and classify (§3.4), combine and annotate them (§3.5) in a way that is more relevant to a software engineer. It is important to stress that the FCA algorithm, filters and analyzers need to be tuned only once, for the specific kind of symptoms that are of interest. Afterwards, this tuning can simply be reused "as is" in order to mine other applications for similar design symptoms.

The novelty of our contribution is not in the idea of applying FCA to source code. What is more important is our particular choice of elements, properties, filters and analyzers, and how these allow us to discover interesting design symptoms in the source code, in a way that is independent of the considered application, and even largely independent of the considered programming language. Proof is that the tool has recently been generalized to support delving Java source code as well.

3.1 Generate FCA elements and properties

Since our goal in this paper is to mine Smalltalk source code, as FCA *elements* we choose source-code entities like classes, methods and method parameters. The reason why we did not (yet) consider additional entities like bundles, protocols and categories, is two fold. Firstly, in order to avoid cluttering the results, we choose to be pragmatic and include initially only the most relevant source code entities. Secondly, we want our approach to be language independent, and most other programming languages do not feature protocols, categories, etc.

As *properties* we take simple substrings of the names of the chosen source-code entities. Evidently, an application's source code contains a wealth of other information, such as call-graph or parse-tree information. Computing such information and reasoning with it, requires vastly more resources, however, which would make our approach much less lightweight. For this very reason, we chose to resort to a simple "name matching" approach at first. Preliminary experiments that reason about similarities in parse trees are ongoing at the moment, but should still be considered future work.

Because we take as properties simple (sub)strings occurring in the names of the considered source-code entities, the discovered concepts will group entities with similar names. Our motivation for choosing these properties, is that in Smalltalk in particular and in many other programming languages, programmers often rely on naming conventions to reveal their intentions and to implement certain programming idioms and design patterns. Keeping the properties simple has the additional benefit that they can be generated and manipulated efficiently.

Nevertheless, in order to limit the number of properties, we do not consider all possible substrings. Instead, we split class, method and parameter names in substrings according to the capitals and other separators occurring in them. In addition, we discard substrings with little conceptual meaning or that are used too often, such as '`with`', '`from`', '`the`', '`object`', as well as substrings that are too small (i.e., less than 3 characters). We also ignore colons, plurals and the difference in case when comparing substrings. For example, the properties associated with a class `QuotedCodeConstant` are the substrings '`quoted`', '`code`' and '`constant`'. The properties corresponding to a method named `#unifyWithDelayedVariable:inEnv:myIndex:hisIndex:inSource:` are '`unify`', '`delayed`', '`variable`', '`index`' and '`source`'.

3.2 Compute the concept lattice

Applying an FCA algorithm to the elements and properties generated in the previous step, results in a large *concept lattice* of several hundreds to thousands of concepts, depending on the size of the application Table 2, Section 5, shows some quantitative results of applying our approach on five different Smalltalk applications.

For now, let it suffice to give an illustrative example of a simple kind of concepts that is discovered by the FCA algorithm: *accessing methods*. Indeed, since both accessor and mutator methods are named after an instance variable, they share the same substring. E.g., the following concept we discovered in one of our case studies groups the `#callStack` accessor and `#callStack:` mutator methods of the `callStack` instance variable defined in the `Environment` class:

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

They are grouped based on the properties 'call' and 'stack' that are shared by these methods, but by no other methods, classes or parameters in the application.

3.3 Filter the concepts

As we will see in Section 5 (see column *#raw* of Table 2), for the considered cases the number of concepts discovered by FCA, before applying any filtering, is of the same order of magnitude as the number of considered elements. This would imply that a software engineer needs to look at a significant number of concepts in order to try and understand the source code. However, the concepts do contain a large amount of redundancy and noise that we can easily filter.

A first filter ignores all concepts that contain two or less elements, since these concepts are generally too small to provide relevant information. Note that this filter discards most *accessing method* concepts, since these typically contain only two elements: an accessor and a mutator method. However, since accessing methods are rather fine-grained, since there are a lot of them, and since they can be inspected with standard browsers easily or they can be retrieved with more dedicated tools, we don't mind that they get discarded.

A second filter ignores all concepts that share only one property (substring). Although this filter may discard some interesting concepts, it does throw away

many more irrelevant concepts. We think that, during initial understanding of an application, getting a quick and focussed idea of certain commonalities in the source-code is more important than getting a precise list of *all* possible commonalities. A nice improvement of this filter would be to discard those concepts of which the properties 'cover' only a small fraction, based on some threshold, of the concept elements' names. As such, the filter becomes relative to the size of the elements' names.

Whereas these two generic filters are independent of the kinds of elements being analyzed, our third filter is more targetted. It discards concepts that contain only classes (with a similar name) in the same hierarchy. These concepts typically do not provide very useful information — since classes belonging to the same hierarchy often have similar names — except if we want to discover exactly which naming convention these classes are relying upon.

3.4 Classify the filtered concepts

Being mere sets of elements (classes and methods) and properties (substrings of the elements' names), the concepts that remain after filtering are rather unstructured. Therefore, we reorganize the concepts automatically in a way that is more easy for a software engineer to analyze and interpret.

More precisely, we flatten the concept lattice and *classify* the concepts in a way that makes more sense to the software engineer. Our classification distinguishes 3 main groups of concepts:

(1) *Single class concepts* group concepts of which all elements are methods (or parameters of those methods) belonging to one and the same class;
(2) *Hierarchy concepts* have a larger scope as they group classes, methods and parameters of those methods, that belong to a single class hierarchy;
(3) For *crosscutting concepts* we explicitly require that at least two different class hierarchies are involved. We do this by verifying that the *most specific* common superclass of the considered classes is `Object` *and* that none of the methods in the concept are defined by the `Object` class itself (which would be a degenerate case of a *hierarchy concept*).

Such a classification helps a software engineer in several ways. Knowing that a certain concept belongs to a given classification helps him to better understand that concept. For example, knowing that a concept containing several methods with exactly the same name belongs to the *hierarchy concepts* classification allows him to qualify those methods as *polymorphic methods*. Or, observing that a *crosscutting concept* contains polymorphic methods that exhibit a lot of duplication, may point out the need for an aspect-oriented solution.

3.5 Combine and annotate concepts

By organizing the concepts in a classification like the above, the structure of the lattice is lost. Concepts that were nearby in the lattice (e.g., that were in a subconcept relationship) will not necessarily belong to the same classification, and vice versa. As a consequence, since there is a lot of overlap between concepts that are nearby in the lattice, when reorganizing the concepts this may lead to redundancy among concepts that get classified into *different* classifications. Whenever possible, however, when nearby concepts in the concept lattice are put in the *same* classification, we automatically reconstruct part of the original structure of the lattice, in order to reduce redundancy. More specifically, we recombine highly overlapping concepts into a single nested one.

In addition, we automatically regroup and annotate the concepts belonging to each classification, in order to present them in a way that is more convenient to the software engineer: different concepts related to the same class(es) are combined, methods are annotated with the classes they belong to, and concepts are annotated with their properties.

4 DelfSTof, our Conceptual Code Mining tool

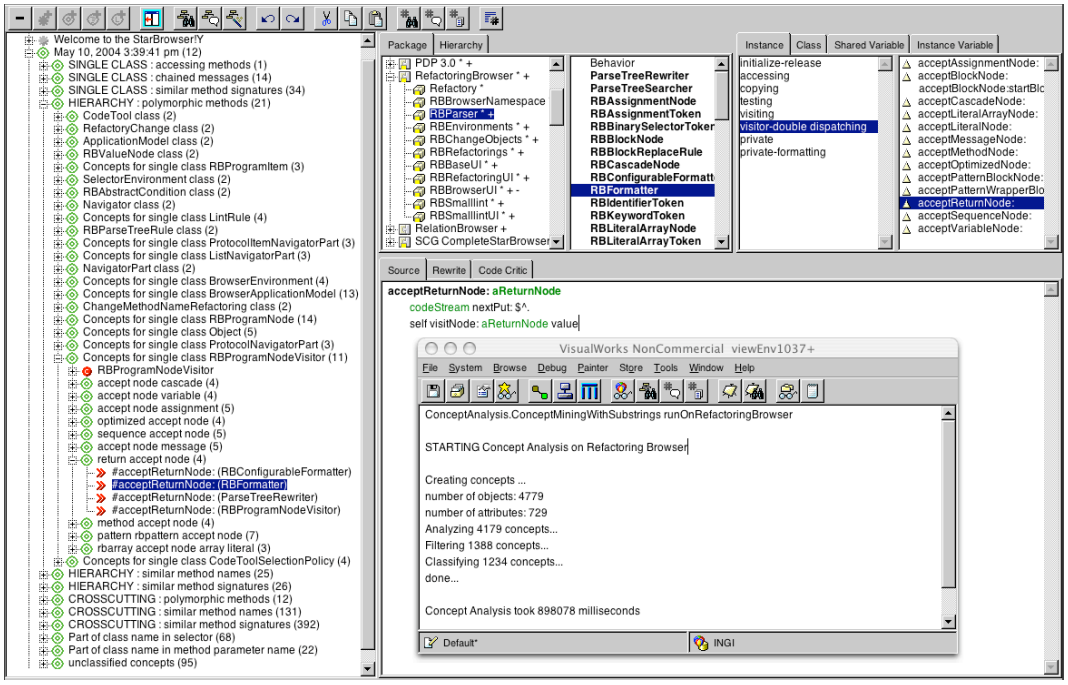

Fig. 2. DelfSTof : Discovered concepts for the Refactoring Browser.

We developed a prototype tool, *DelfSTof* [2], that implements the approach outlined above, and presents the discovered concepts in a way that is easy to use and manipulate. We capitalize the letters "ST" because the tool is implemented completely in Smalltalk and originally only analyzed Smalltalk source code.

The tools consists of an efficient FCA algorithm, a set of filters, and a set of 'analyzers' that are in charge of the classification, combination and annotation of concepts. The resulting classification of concepts is directly visualized with the *StarBrowser* [4]. A screenshot of the tool, which is essentially a StarBrowser plugin, is presented in Figure 2.

5 Discovered Design Symptoms

We applied our tool to five different cases, as summarized by Table 2. Every row in this table corresponds to a different application of which we delved the source code for design symptoms. *SOUL* is an interpreter for a Prolog-like language and *DelfSTof* is our own conceptual code mining tool. We chose these two applications because we know their implementation well, which allowed us to better assess the relevance of the discovered results. Both the *StarBrowser* and the *Refactoring Browser* are advanced Smalltalk browsers and *CodeCrawler* is a language-independent reverse engineering tool which combines metrics and software visualization. These cases were chosen because they are relatively stable and well-developed, have been around for several years, and are of perfect size for initial experimentation.

Table 2
Quantitative results of FCA applied to some Smalltalk applications.

Case	#elements	#properties	#raw	#filtered	time (sec)
DelfSTof	756 (135)	237	617	126	5
StarBrowser	731 (52)	352	740	115	7
SOUL	1469 (111)	434	1188	281	22
CodeCrawler	1370 (93)	477	1419	327	24
Refactoring Browser	4779 (271)	729	4179	1234	414

[2] The name coins the Dutch word 'delfstof', which designates the result of a delving process. In English, the first meaning of the verb "to delve" is "to make careful investigation for facts and knowledge". Coincidentally, the pronunciation of the word "delfstof" sounds like the English "delve stuff" which is indeed what the tool does.

The column *#elements* gives an indication of the size of each considered case as it equals the total number of classes plus methods in that case. (The number between brackets is the number of classes.) Note that we did not consider method parameters as separate elements since they are part of the method signature.

The column *#properties* shows the number of substrings that have been generated from the considered elements. We observe that the number of properties is always a factor 2 to 4, in the case of the Refactoring Browser even almost a factor 7, *less* than the number of elements. This is a good sign as it implies that a significant amount of the properties are actually shared by the elements.

The third column *#raw* shows the raw number of concepts discovered by FCA. Column *#filtered* shows how many concepts remain after having applied the simple filters that were explained in §3.3. We observe that, after filtering, there remain about 4 to 7 times less concepts than the number of considered elements. We still think that this is a bit too much, especially for larger cases, but we will come back to this discussion in Section 7.

For all considered cases, the time of computation — which includes all steps explained in Section 3 and not only the computation of the concept lattice — was acceptable (ranging from a few seconds to a few minutes), although theoretically it increases in a non-linear way with the number of considered objects.

The remainder of this section discusses some of the "design symptoms" we discovered when manually analyzing the results of our FCA experiments. As a matter of fact, we could refine the classification of §3.4 to classify explicitly and automatically most of these symptoms as well (for example, by using our logic meta-programming environment SOUL). However, a certain trade-off needs to be made, since this would require more automated analysis and thus slow down the tool. Also, we want to keep the tool sufficiently general so that it still can be applied to other languages (maybe even non OO languages). Nevertheless, we already extended our tool so that it classifies automatically the programming idioms listed in Subsection §5.1 : *polymorphic methods*, *chained messages* and *delegating methods*.

5.1 Programming idioms

Polymorphic methods are a symptom of good design that is readily recognized by our tool, since polymorphic methods have exactly the same name. Consequently, they are grouped together in a concept. In addition, if there are several polymorphic methods for a same class hierarchy, these will all be grouped together automatically in a single combined concept.

For example, in the source code of the *Refactoring Browser* we discovered a concept containing 4 methods named `#acceptReturnNode`, implemented on different subclasses of `RBProgramNodeVisitor`. This is a typical example of polymorphism of which many examples can be found in any OO application. In fact, for the class `RBProgramNodeVisitor` alone many more polymorphic method concepts were discovered, as can be seen from Figure 2.

Polymorphic methods across class hierarchies are equally interesting to detect as they may trigger interesting refactorings or tell us something about possible multiple inheritance problems the original developers encountered. We will come back to this in Section 5.5

Chained messages are concepts that group a method together with some of its auxiliary methods in the same class. These chains are recognized by FCA since the auxiliary methods often have a name that is similar to that of the originating method, though sometimes slightly longer and taking an extra parameter. E.g., in the *CodeCrawler* application the class `CCMetricsChooserDialog` implements a method `#applyChosenMetrics`, which calls an auxiliary method `#detectChosenMetrics`, which in turns calls `#assignChosenMetricsTo:`. These 3 methods share the substrings 'chosen' and 'metrics'.

Delegating methods delegate responsibility by calling a method with the same name. Our tool discovered some interesting sets of delegating methods with a similar name that all belonged to the same class. The presence of many such delegating methods in a single class may indicate that the class is a *Decorator* [5].

5.2 Code duplication

By closely inspecting the discovered concepts, we also detected several cases of copy and paste reuse: several concepts contain methods that not only have a similar name, but a similar implementation as well. This may seem logical, since methods that implement similar behavior can be expected to have similar names. However, from an implementation point of view, this duplicated code could just as well be factored out and reused.

For example, in *CodeCrawler*'s `CEVModelHistory` class we discovered a concept with 2 methods `#predecessorModelNameOfModel:` and `#predecessorModelNameOfModelNamed:` which had nearly the same implementation. Since one of them is no longer being used, we assume that the original developer(s) created one of the methods by copying it from the other one, then replaced all calls to the old one to the new one, but in the end forgot to remove the old method.

In general, particular reasons for duplication may become clear by looking at the classification of the concepts:

- a developer who was not aware that a method implementing the desired behavior was already defined, may accidentally implement a method with a similar signature and behavior. Such methods are often grouped in concepts classified as *hierarchy concepts*, because the method that already implements the desired behavior is probably not implemented by the class the developer is looking at, but by one of its sub- or superclasses.
- a method was needed whose behavior differed slightly from the behavior already implemented by an existing method, and this behavior was copied and adapted slightly, without extracting the common code into a separate method (or merely deleting the original version if it is no longer needed, such as in the example of the `CEVModelHistory` class above). Concepts containing such duplicated methods tend to be classified as *single class concepts*, because such duplication typically occurs inside a single class.
- the duplicated behavior could not be factored out into a single piece of code, and thus could not be reused. This is mostly due to the fact that the duplicated code occurs in classes defined in different class hierarchies. As such, these concepts are often classified as *crosscutting concepts*.

5.3 Design patterns

As many design patterns [5] use certain naming conventions, it is no surprise that they are detected by our tool. For example, the *Visitor* pattern uses the convention that each *visit* method defined by a *visitor* class encodes the name of the class being visited. Since they clearly share some substrings, our tool will group a class and its corresponding visit methods inside a single concept.

The *Refactoring Browser* uses a *Visitor* design pattern in order to perform a variety of operations on source code entities, such as renaming and moving them. These entities are represented as subclasses of the `RBProgramNode` hierarchy, while the visitor hierarchy is defined by the `RBProgramNodeVisitor` class hierarchy. Our tool recovered this design pattern instance in two concepts:

(1) A combined concept in classification *hierarchy concepts* which groups all *polymorphic methods* in the `RBProgramNodeVisitor` hierarchy, that implement the behavior to be executed when a particular term is visited. The concept consists of a several sub-concepts, each of which contains all methods defined by subclasses of class `RBProgramNodeVisitor`, dealing with one particular term. For example, one such concept is defined by the properties 'accept', 'return' and 'node', and contains various implementations of the `acceptReturnNode:` method, defined in the `RBProgramNodeVisitor` hierarchy and

responsible for implementing behavior associated to a `RBReturnNode` object. More specifically, the concept consists of four `acceptReturnNode:` methods, implemented in the classes `RBFormatter`, `RBConfigurableFormatter`, `ParseTreeRewriter` and `RBProgramNodeVisitor`.

(2) The second concept is also a *hierarchy concept* and contains all `acceptVisitor:` methods defined by subclasses of `RBProgramNode`. These methods are responsible for calling the appropriate method, corresponding to the node being visited, in the supplied visitor object. They are grouped based on the 'visitor', 'accept', 'program' and'node' substring properties. This is an example of a concept that takes into account both the method's name and the name of its formal parameter, since the `acceptVisitor:` methods always define a formal parameter named `aProgramNodeVisitor`.

Note that the *polymorphic methods* depicted in Figure 2 are also a part of a visitor pattern, used in the *Refactoring Browser* implementation.

Another example our tool detected is the *Abstract Factory* design pattern that is used in the *Soul* application. The factory is responsible for creating new instances of many different classes in the system, among others, subclasses defined in the `AbstractTerm` and `HornClause` hierarchies. Its presence was recognized 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. In addition, the choice of the word 'make' strengthens our belief that it indeed is a factory.

Several other design patterns, such as the *Builder*, *Observer* and *Decorator* design patterns, were detected in other applications as well.

5.4 Relevant domain concepts

Frequently occurring properties give a good idea of what the important concepts in the application or problem domain are. This information is very useful to understand the domain, and to provide a common vocabulary which can be used to talk with maintainers. For example when applying our FCA tool to the source code of *DelfSTof* itself, we found several concepts with properties like 'concept', 'attribute', 'analyze(r)', 'filter' or 'classification'. Likewise, we found concepts with properties 'lint rule' and 'browser' in the *Refactoring Browser*, as well as concepts whose properties name the different refactorings, such as 'add method', 'rename variable' and 'change name space'. In *SOUL*,

we found concepts with properties 'resolution' and 'repository'. Clearly, these strings convey information that is important in the domain of the respective applications.

5.5 Opportunities for refactoring

In addition to revealing interesting symptoms of good design and important domain concepts, our approach can identify opportunities for refactorings [6] that improve the source code quality.

An obvious opportunity for refactoring is to get rid of some of the code duplication that was detected (§5.2). The way a concept containing duplicated methods is classified, can provide useful hints about which refactorings to apply. If the concept is classified as a *single class concept*, the duplication occurs in a single class, and an *Extract method* refactoring is appropriate. If the concept occurs in the *hierarchy concepts* classification, a combination of *Extract method* and *Pull up method* refactorings is probably more appropriate. Of course, if the duplication is caused by having copied a method that is no longer used, it suffices to simply remove that method.

Concepts that were recognized as *polymorphic methods* can also be inspected for refactoring opportunities, to ensure that polymorphism is well-implemented:

- We could check whether all classes in a class hierarchy understand the polymorphic method. If not, we may need to add an additional one.
- A polymorphic method might also be implemented by several subclasses of a particular superclass, but not by that superclass itself. In that case, a *Pull up method* or *Add class* refactoring may be appropriate to define the methods in the superclass, or to insert an intermediate superclass.

  A particular example we discovered in *SOUL* is the `#updateRepositories` method, which is only defined separately in subclasses `RepositoryBrowser`, `SoulQueryBrowser` and `SoulClauseBrowser` but not in their common superclass `ApplicationModel`. Introducing an intermediate superclass here might be appropriate.

  In the *CodeCrawler* case, we also found several examples where the *same* polymorphic method appeared in several subclasses, but not in their common superclass. However, in most of these examples the code in one subhierarchy did not seem to be used, which made us believe that the code was moved from one part in the hierarchy to another, but that the developer(s) forgot to remove the original code.
- *Crosscutting polymorphic methods* are also suspicious. For example, we discovered that in *SOUL*, the `#usesPredicate:multiplicity:` method is implemented in both the `AbstractTerm` and `HornClause` hierarchies. Smalltalk, which has dynamic typing, still allows classes of these hierarchies to be used polymor-

phically. A static type system, as in Java, would prohibit such polymorphic use. In that case, an interface probably has to be introduced in order to avoid explicit type checks and type casts. If the crosscutting polymorphic methods also happen to exhibit a lot of code duplication, the solution might be to introduce aspects into the application. In that respect, our tool might also be considered as an aspect mining tool, capable of detecting aspect opportunities. Future research into this area appears very interesting.

Concepts that represent design pattern instances can also be scanned for particular design flaws. For example, the *Visitor* design pattern requires that each visitor class defines an appropriate *visit* method and, vice versa, that each element class defines an *accept* method that calls the appropriate *visit* method. The concepts that identify instances of the *Visitor* design pattern can be used to inspect the implementation in a quick and straightforward way, and verify whether these constraints are adhered to.

6 Related Work

The use of FCA in software engineering is not new. Snelting et al. [7] use FCA to reengineer C++ class hierarchies, while Arévalo et al. [3] analyze object-oriented framework reuse using FCA. Closer to our work are the techniques by Tonella et al. [8] and Dekel et al. [9]. The former use FCA to detect instances of design patterns in source code. Since they specifically tune the FCA algorithm for detecting such instances, they are not able to detect other kinds of design symptoms, as our approach does. The latter use FCA to reveal the structure of single classes only. They partition the methods of a class, according to the fields these methods use, and then use the concept lattice to visualize and understand the structure of that class. Tilley et al. [10] provide an overview of the use of FCA for several other software engineering purposes.

A large number of tools to verify the quality of the source code of an application exists. The spectrum ranges from very simple tools that detect basic coding errors [11], over specialized clone detection tools [12,13,14,15], to tools that detect high-level bad smells [16,17,18] and propose appropriate refactorings.

Other tools exist that are capable of detecting high-level structures in source code, such as coding conventions and design patterns [19,20,21,22]. The main difference between these tools and ours, is that our approach requires no a priori knowledge. Most of these existing tools, however, rely on the fact that design pattern implementations follow *particular* naming conventions and guidelines. Our approach is not targeted to detecting a specific kind of conventions, but is able to detect a variety of symptoms that reveal bad design (bad smells,

duplication), good design (design patterns, programming idioms), and opportunities for refactoring. This is particularly useful during initial understanding, whereas in a later phase, when the code is better understood, a more directed tool is preferable.

Research in the domain of aspect mining is also related. Tonella et al. [23] and Breu et al. [24] find aspect instances in existing applications by means of dynamic execution traces. The former work even uses formal concept analysis to that extent. Our approach seems to complement these approaches, since we combine static analysis techniques with formal concept analysis. Marin et al. [25] use the fan-in metric in order to mine for aspects, whereas Shepherd et al. [26] and Bruntink et al. [27] use clone detection techniques.

7 Future work

An important topic of future work is to further *improve the filtering* of the concepts discovered by FCA, so as to reduce the remaining redundancy in the discovered concepts. The problem is that this redundancy occurs between concepts that are classified in different categories. As was briefly mentioned in §3.3, this redundancy is a consequence of having flattened the concept lattice. By doing so we lost some important dependencies between concepts. But exactly this information may be useful to get rid of the redundancy. Therefore, we propose to keep the lattice as an internal representation, so that advanced filters can take advantage of it to resolve the remaining redundancy.

Since the time of computation of our tool theoretically increases in a non-linear way with the number of considered objects, this may pose problems regarding the scalability of the approach. However, we do not think that it is a good idea to apply the approach to *very* large amounts of source code, since the tool assumes that certain naming convention are adhered to in a consistent way. This is unlikely for very large cases. In such a context it would be better to apply the approach multiple times on several smaller pieces of which we know they are more or less independent and have been developed by a same development team. As a side-effect, this will also resolve the problem with the time of computation.

8 Conclusion

In this paper we proposed a fairly efficient tool, capable of delving an application's source code for meaningful and interesting symptoms of good and bad

design. The tool combines formal concept analysis with filtering and classification techniques, in order to provide simple and effective views on structural commonalities in the source code. In order to validate the approach we applied the tool on a number of small to medium-sized Smalltalk applications. In spite of relying on nothing more than similarity of names of source-code entities, we discovered symptoms of programming idioms, code duplication, design patterns and domain concepts, as well as interesting opportunities for refactoring. Although the approach can still be improved in many ways, in particular to further reduce the redundancy, we do believe it can be very useful when an application has to be understood, little a priori knowledge about the source code is available and time is scarce.

9 Acknowledgements

We thank T. Mens, R. Wuyts, R. Riquelme, S. González and G. Arévalo as well as the anonymous reviewers of the ECOOP'2004 *workshop on object-oriented reengineering* and the AOSD'2004 workshop on *foundations of aspect languages* for their feedback on earlier versions of this paper. We thank F. Spiessens for his efficient implementation of the FCA algorithm in Smalltalk. Finally, we thank the ESUG'2004 reviewers and our shepherd Serge Demeyer in particular, for their useful comments.

References

[1] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufmann and DPunkt, 2002.

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

[3] G. Arévalo, T. Mens, Analysing object-oriented application frameworks using concept analysis, in: Proceedings of the Inheritance Workshop at the European Conference on Object-Oriented Programming (ECOOP), 2002.

[4] R. Wuyts, S. Ducasse, Unanticipated Integration of Development Tools using the Classification Model, Journal of Computer Languages, Systems and Structures 30 (2003) pp. 63–77.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Massachusetts, 1994.

[6] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] G. Snelting, F. Tip, Reengineering Class Hierarchies Using Concept Analysis, in: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1998.

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

[9] U. Dekel, Y. Gil, Revealing Class Structure with Concept Lattices, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), 2003.

[10] T. Tilley, R. Cole, P. Becker, P. Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities, in: Proceedings of the International Conference on Formal Concept Analysis, 2003.

[11] S. Paul, A. Prakash, A Framework for Source Code Search using Program Patterns, IEEE Transactions on Software Engineering 20 (6).

[12] B. S. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE), 1995, pp. 86–95, received IEEE Outstanding Paper Award.

[13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society Press, 1998, pp. 368–377.

[14] S. Ducasse, M. Rieger, S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, in: H. Yang, L. White (Eds.), Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society Press, 1999, pp. 109–118.

[15] R. Komondoor, S. Horwitz, Using Slicing to Identify Duplication in Source Code, in: Proceedings of the International Symposium on Static Analysis, Springer-Verlag, 2001.

[16] Y. Kataoka, M. D. Ernst, W. G. Griswold, D. Notkin, Automated Support for Program Refactoring using Invariants, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2001, pp. 736–743.

[17] T. Tourwé, T. Mens, Identifying Refactoring Opportunities Using Logic Meta Programming, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, Benvento, Italy, 2003, pp. 91 – 100.

[18] E. van Emden, L. Moonen, Java Quality Assurance by Detecting Code Smells, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, 2002.

[19] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien, Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together, in: Proceedings of the International Conference on Automated Software Engineering (ASE), 2001.

[20] K. Brown, Design reverse-engineering and automated design pattern detection in smalltalk, Master's thesis, University of Illinois at Urbana Champaign (1996).

[21] K. Mens, I. Michiels, R. Wuyts, Supporting Software Development through Declaratively Codified Programming Patterns, Journal on Expert Systems with Applications .

[22] R. Wuyts, Declarative Reasoning about the Structure of Object-Oriented Systems, in: Proceedings of TOOLS USA'98, IEEE Computer Society Press, 1998, pp. 112–124.

[23] P. Tonella, M. Ceccato, Aspect Mining through the Formal Concept Analysis of Execution Traces, in: Proceedings of the Working Conference on Reverse-Engineering (WCRE), 2004.

[24] S. Breu, J. Krinke, Aspect Mining Using Dynamic Analysis, in: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik, Vol. 23, Bad Honnef, Germany, 2003, pp. 21–22.

[25] M. Marin, A. van Deursen, L. Moonen, Identifying Aspects using Fan-In Analysis, in: Proceedings of the Working Conference on Reverse-Engineering (WCRE), 2004.

[26] D. Sheperd, E. Gibson, L. Pollock, Automated mining of desirable aspects, Tech. Rep. 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716 (2004).

[27] M. Bruntink, A. v. Deursen, R. v. Engelen, T. Tourwé, An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society, 2004.