A Catalog of Patterns for Concept Lattice Interpretation in Software Reengineering

Muhammad Usman Bhatti, Nicolas Anquetil, Marianne Huchard, Stéphane Ducasse

A Catalog of Patterns for Concept Lattice Interpretation in Software Reengineering

Muhammad U.Bhatti*, Nicolas Anquetil*, Marianne Huchard†, and Stéphane Ducasse*
*RMoD Project-Team INRIA - Lille Nord Europe USTL - CNRS UMR 8022, Lille, France
Email: {firstname.lastname}@inria.fr
†LIRMM, CNRS and Université de Montpellier-2, Montpellier, cedex 5, France
huchard@lirmm.fr

***Abstract*—Formal Concept Analysis (FCA) provides an important approach in software reengineering for software understanding, design anomalies detection and correction. However, FCA-based approaches have two problems: (i) they produce lattices that must be interpreted by the user according to his/her understanding of the technique and different elements of the graph; and, (ii) the lattice can rapidly become so big that one is overwhelmed by the mass of information and possibilities. In this paper, we present a catalogue of important patterns in concept lattices, which can allow automating the task of lattice interpretation. The approach helps the reengineer to concentrate on the task of reengineering rather than understanding a complex lattice. We provide interpretation of these patterns in a generalized manner and illustrate them on various contexts constructed from program information of different open-source systems. We also present a tool that allows automated extraction of the patterns from concept lattices.**

I. Introduction

Formal Concept Analysis (FCA) is a mathematical technique to discover significant groupings of objects having similar attributes [14]. FCA can be applied to program entities, which helps in generating high-level views of a program for program understanding and identifying and correcting some anomalies in software systems. For example, FCA is used to identify objects in procedural code by looking at functions accessing global variables [26], [9]. Equally, it is applied to object-oriented systems by analyzing software components (*e.g.,*, classes, attributes or methods) and their relationships (*e.g.,*, belongs-to, uses or defines). Some studies aim at understanding object-oriented systems [3], [8], [23], others use program information for reengineering classes [28], [29], [5]. The extracted program information is used to construct concept lattices that are presented to the reengineers for analysis. The reengineer then explores a lattice to uncover important information.

The problem with the FCA techniques is that they produce concept lattices that are not readily comprehensible for developers who need to spend a good amount of effort to interpret these lattices. A possible solution is to break the information being fed into FCA into small chunks [5]. However, the approach is not scalable as the chunks need to be individually analyzed, which requires generating several lattices. Another issue is that concept lattices extract many more concepts than the number of objects they are given as input [1], thus adding complexity over the semantic complexity of the lattice.

The objective of the paper is to reduce concept lattices to a few interesting node and subgraph patterns such that the user does not have to interpret each node and its relationships nor to analyze the entire lattice in detail. We don't pretend that all useful findings are captured by these patterns, but they simplify the task of lattice analysis by proposing some accepted semantics. We look for the lattice patterns in an automated way, which could open the path for semi-automated reengineering actions.

In this paper, we define the patterns that proved to be useful in many cases of FCA applied to software engineering domain. We provide an interpretation in a generalized way, explaining their interest. Later, we exemplify them using different information from various open-source systems. These patterns are implemented in a prototype and we give some figures on the reduction in information to process for a concrete example.

This paper is organized as follows: Section II presents existing work that uses FCA in software reengineering and motivates our approach. Section III presents nodes and subgraphs patterns. Section IV proposes a validation on open-source systems and Section V describes the prototype for pattern identification. Section VI concludes the paper.

II. Motivating a generic interpretation tool for concept lattices

In this section, we present different studies that have been performed to understand and restructure software systems through FCA. Then we provide motivation for our proposed approach.

A. Existing work

a) Module and Object Identification with FCA: Sahraoui *et al.* [26] present a method that extracts objects from procedural code using FCA. Important concepts are looked for in the resultant lattices using heuristics. Another approach compares the object identification capability of clustering and FCA techniques [9]. A few heuristics are described to interpret the concepts of the lattice. An approach to transform a COBOL legacy system to a distributed component system is proposed by Canfora *et al.* [6]. Siff and Reps explore the relationship between program variables and procedures through FCA for restructuring of C programs [27].

b) Object-Oriented Reengineering and FCA: Godin *et al.* [15] proposed in 1993 to analyze classes by their member methods to evaluate and refactor OO class hierarchies using FCA. Dicky *et al.* [10] define an incremental algorithm and study the method specialization in the FCA framework. Falleri *et al.* [13] compare several FCA configurations [7]. Leblanc *et al.* [17] apply FCA to extract Java interface hierarchies from Java classes.

Snelting and Tip [28] present a framework for detecting and remediating design problems in a class hierarchy; it is used in [29] for refactoring Java class hierarchies. Arévalo, Ducasse and Nierstrasz [3] use FCA to understand how methods and classes in an object-oriented inheritance hierarchy are coupled by means of the inheritance and interfaces relationships. Lienhard, Ducasse and Arévalo [21] use FCA to identify traits in inheritance hierarchies. Dekel and Gil [8] use FCA to visualize the structure of Java classes and to select an effective order for reading the methods. In [5] a tool-assisted technique is presented to identify useful abstractions and class hierarchies in procedural object-oriented code.

FCA is used to detect the presence of bad smells and design patterns [2], and to suggest appropriate refactorings to correct certain design defects [24]. FCA is also used to understand a system's source code for relevant concepts of interest [23] and to cluster words for feature location [25].

FCA is also used to examine information generated from program execution to reconstruct control flow graphs [4] and feature location [12]. FCA is used in aspect mining [20], detection of design-level class attributes from code [30], and searching for code patterns and their violations [22].

c) FCA Filtering: Some studies, applying FCA to software reengineering, have suggested concept filtering techniques to resolve the problem of lattice complexity. The notion of concept partitions is used to filter the concepts that are not interesting for creating modules [27]. Mens *et al.* filter concepts according to some properties[1] of the concepts in the lattices constructed by their approach [23]. Arévalo *et al.* describe a few heuristics to reduce the search for concepts that describe important class hierarchy schemas [2]. Joshi and Joshi [19] provide a few patterns that may emerge when context is based upon methods and attributes of a class.

There is a plethora of FCA-based techniques that aim to analyze software systems for reengineering purposes. The diversity of these approaches shows the interest in FCA as a tool to certain kinds of software analysis. Some approaches apply context-specific heuristics for filtering non-interesting concepts. However, these heuristics remain tied to a specific context and cannot be generalized. Therefore, there is a need to define a unifying framework for lattice interpretation. This paper goes in that direction by describing a few patterns of nodes and subgraphs in concept lattices.

B. Synthetic view of FCA in Software Reengineering

One of the strengths of FCA for program comprehension and software reengineering is the wide range of contexts that can be used. For each different context, the method provides different insights into reengineering opportunities.

OO systems consist of different entities such as packages, classes, methods, attributes, or variables. In Table I we summarized the main entities and relationships that could be used as context (other ones could be invented). In the table, rows are formal objects, and columns formal attributes. In each cell of the upper half, a blank means there is no possible relationship (we found no interesting relationship to link a method formal object to packages formal attributes), U stands for use (a method uses another method by calling it, or a class uses another class through inheritance), and C stands for contains (or declare, a class contains an identifier, a method contains a variable). The relationships in boldface (upper half) denote contexts that we found already studied in the literature. In that case, a representative reference is given in the lower half of the table. The "×"s in the lower half denote possible contexts that we identified and for which we know of no prior published research. They represent unexplored areas of FCA and software reengineering research.

TABLE I
MAIN POSSIBLE CONTEXTS FOR FCA; FORMAL OBJECTS ARE IN ROWS, FORMAL ATTRIBUTES ARE IN COLUMNS, RELATIONSHIPS ARE USE AND CONTAIN; IN LOWER HALF, REFERENCES INDICATE THE CONTEXTS THAT WERE ALREADY STUDIED (ALSO IN BOLD FACE IN UPPER HALF) AND "×"S MARK POSSIBLE CONTEXTS THAT HAVE NOT BEEN EXPLORED YET

		Formal attributes				
		pckg	class	meth.	att.	var.
Formal objects	pckg	C,U	C,U	C,U	C,U	C,U
	class	U	C,**U**	**C**,U	**C**,U	C,U
	meth.		U	**U**	**U**	C,U
	att.		U			
	var.		U	**U**	**U**	U
	pckg	×	×	×	×	×
	class	×	[5], [32]	[21], [15] [18]	[13]	×
	meth.		×	[3], [5] [22]	[3], [8] [30]	×
	att.		×			
	var.		×	[28]	[28]	×

Table I considers only the main formal objects and formal attributes in software reengineering for OO systems. A comprehensive survey of the contexts studied in software engineering is provided by Tilley *et al.* [31].

C. Concept lattice interpretation

FCA is a flexible technique that may be used on a wide range of contexts. The number of unexplored contexts, even for the simple enumeration of possibilities proposed in the Section II-B, gives an idea of its potential. However before it gains larger use, we believe two problems need to be solved.

First most of the existing approaches, leave the analysis work to the user. Concept lattices are complex abstract constructs that may prove difficult to interpret. Sahraoui *et al.* [26] recognized this problem and proposed a tool to help analyzing the lattices.

Another issue with FCA is the size of the lattices they produce [1]. This again points toward the need to provide an

[1] size of the concept's "intent" and "extent".

assisted solution for lattice analysis. Some filtering techniques exist in the literature to remove unwanted concepts from lattices. However, these techniques are dependent upon their contexts because these are geared towards unrevealing specific code patterns. The solution of decomposing a context into smaller chunks requires a lot of effort to generate lattices representing each chunk. A natural solution lies in automating the interpretation of lattices. This automation will be more useful if it can be applied indifferently to lattices built on any formal context. The user would not be required to analyze each node in the lattice; he will search for patterns in the lattices and useful nodes will be identified without spending too much time on the lattice. Moreover, an automated technique can extract these patterns without requiring the user to understand the complexities of FCA. We hope to define a dictionary of such patterns that along with a definition of the meaning of a concept for each possible context, could propose to users an interpretation for any lattice built from a known context.

In the next section, we present a catalogue of patterns that represent interesting constructs in concept lattices from the software engineer point of view. These patterns help the user in two ways. First, they help to reduce the work of understanding a complex lattice for interpreting a few node and graph patterns. Hence, the work is reduced to look for these patterns and understand their interpretation. Second, because we consider generic subgraph patterns, a tool can be built to extract these patterns from the lattices, which will greatly simplify the work of analyzing these lattices.

III. Patterns in Concept Lattices

In this section, we present concept lattice patterns intended to help software engineers analyze the result of FCA. These patterns are sufficiently generic to be applied to a large set of possible contexts.

A. Nodes in Concept Lattices

The patterns we will present depend on typical arrangements of nodes and vertices, but also on the specific nodes that concept lattices may/should contain. Therefore, before going to the patterns, we will take a look at the four types of nodes that a concept lattice may contain. In the following, we borrow from the Conexp tool the way these nodes are displayed. The tool displays each node as a circle. A black lower semi-circle in the node indicates the presence of formal objects introduced by the node (*i.e.,* that no sub-concept has). A grey upper semi-circle in the node indicates the presence of formal attributes introduced by the node (*i.e.,* that no super-concept has). Four types of nodes are possible :

- **Full (black and grey):** The node introduces formal attributes that its super-concepts don't have and has formal objects that its sub-concepts don't have.
- **Grey:** The node introduces formal attributes that its super-concepts don't have, but all its formal objects (if it has any) appear in a sub-concept. Such node must have more than one direct sub-concept.
- **Black:** The node has formal objects of its own (that no sub-concept has), but "inherits" all its formal attributes (if it has any) from its super-concepts. Such node must have more than one direct super-concept.
- **Empty:** The node has no formal attribute or object of its own. It must have more than one direct super-concept and more than one direct sub-concept.

B. A Catalogue of Patterns

We now describe some subgraph patterns that we identified as useful to help analyze a concept lattice. The criteria that we used to choose these patterns are:

- They have a clear meaning that can be interpreted for any formal context; Hence, the patterns are *generic* enough to extract meaningful information;
- They represent important groups of formal objects and formal attributes, *useful* for the user;
- They produce very few false positives: If the patterns identify too many false-positives in the results, the user is required to look at the lattice to filter the false-positives. Hence, the effectiveness of the patterns would be lost. One can *rely on* the patterns and not analyze the lattice. We prefer limiting the false positives at the expense of having many false negatives.

The patterns are defined as specific topology of nodes and edges, sometimes accompanied by a specific type of node for one or more of the members of the pattern. Some of these patterns are illustrated in Figure 1 to help the reader visualize them.

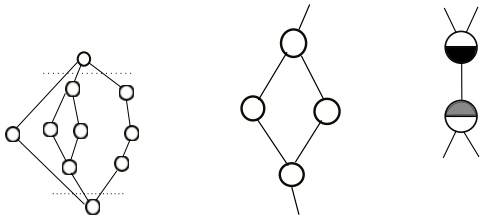

Fig. 1. Some of the patterns in concept lattices

1) Top node ($\top$): We identified two cases of interest for the top node:

Top-black The pattern reveals the fact that the formal objects attached to the top node don't have relationship to any of the formal attributes present in the context. It indicates formal objects that are not relevant to the context being studied.

Top-grey or Top-full The existence of a grey or full node at the top of a lattice marks a set of formal attributes (the grey part) that are in relationship to all the formal objects contained in the context. The pattern represents an important set of formal attributes that form the very basis of the context.

2) Bottom node ($\bot$): We may identify two cases:

Bottom-grey The pattern reveals the fact that the formal attributes attached to the node are not in relationship with any of the formal objects present in the context. Likewise, the

pattern is important as it depicts attributes that are not relevant to the context being studied.

Bottom-black or Bottom-full The existence of the black or full node at the bottom of a lattice illustrates a set of formal objects that are related to all the formal attributes present in the formal context.

3) Horizontal Decomposition: Horizontal decomposition (see Figure 1(a)) is a pattern that appears when one can identify disjoint subgraphs when removing the top and bottom of the lattice. In the illustrating figure, there are three disjoint subgraphs. Horizontal decomposition points to the presence of disjoint sets of relationships between formal objects and formal attributes. This is somehow similar to having several disjoint contexts. The different subgraphs may actually have formal attributes or formal objects in common in the case of Top-grey/Top-full and Bottom-black/Bottom-full, but assuming we ignore these (see discussion above), then each subgraph may be considered independently from all the others. Snelting and Tip mention horizontal decomposition in [28]. A less constrained pattern would be to find a horizontal decomposition, with non trivial subgraphs, between two nodes that are not the top or the bottom of the lattice. This is also related to the Module pattern that will be discussed immediately.

4) Module: In partially ordered set theory, a module represents a group of nodes that are connected to the rest of the graph through a unique top node and a unique bottom node [16]. The supremum and infimum of the module are part of it. Figure 1(b) illustrates a simple example of module in a concept lattice. A module represents an important pattern because it can be considered as a sublattice inside a concept lattice. One could imagine collapsing the entire module into one "composite node" without changing the semantics of the lattice (just make it a bit more abstract). Also, all the patterns applicable to the concept lattice can also be applied to the module. The module in itself can be seen as a smaller individual lattice that can be analyzed independently of the rest.

5) Irreducible specialization: Specializations are the basis of a lattice and one finds them everywhere: Every arc in a concept lattice marks a specialization relationship. Yet the pattern is interesting when the specialization occurs between a black (or full) super-concept and a grey (or full) sub-concept (illustrated in Figure 1(c)). Irreducible specialization patterns depicts two nodes that should not be merged. The upper node (in Figure 1(c)) needs to exist because it has its own formal object(s) and the lower node needs to exist because it has its own formal attribute(s). That is to the say specialization pattern represents two entities in a system that are relevant in the system they belong to. When the super-concept (sub-concept) in the specialization pattern is the top node (resp. bottom node), we further require that it is a full node, to ensure that it represents a complete concept with formal objects and formal attributes. When we are not dealing with the top or bottom nodes in the lattice, we don't need this restriction because the nodes in the pattern can inherit formal attributes (formal objects) from their super-concepts (resp. sub concepts).

IV. Concrete Examples

In this section, we present a validation of these patterns to evaluate that the patterns do appear in the concept lattices, and these reveal important information about the underlying program. For the examples, we experiment with various formal contexts from different program entities of open-source systems. These contexts are similar to existing contexts of the literature. The contexts will be presented as triplet: O-R-A that is to say: formal Object, Relation, and formal Attribute. Program information for the lattices is extracted through FCAParser[2].

A. Member use

This formal context is based on the following information:

- O = All methods within a class except getters and setters
- A = Attributes of the class
- R = The method accesses an attribute directly or through a getter method

The concepts resulting from this context represent different features that a class may implement, assuming that each feature will be composed of methods accessing a particular set of attributes. The example used will be the class Main of JMemorize[3]. Figure 2 shows the lattice resulting from this example. We may identify the following patterns in the concept lattice:

- **Top black:** The two methods in the top-black node do not access any attribute of the class. Code analysis shows that copyFile is a utility method whereas onProgramEnd is an empty method in the class.
- **Horizontal Decomposition:** Removing the top and bottom nodes of the lattice, we are left with 6 independent subgraphs: (i) a set of three disjoint subgraphs on the far right having methods main, run, and startStats, (ii) the bulk of the nodes in the centre, (iii) one node with methods exit, rmPrgObs and addPrgsObs on the lower left, and (iv) two nodes with methods clrThrw, logThrw. The subgraphs identify independent concerns of the class (convertible to traits). For example, the log (on the far left) or the handling of observers when the program ends (lower left).
- **Irreducible specialization:** There is one instance of this pattern on the left of the lattice. As explained, it indicates that we have here two features that could not be easily merged.
- **Module:** The same two nodes on the far left illustrate a case of the simplest possible module. The two nodes could be grouped in one to simplify the lattice. This is not incompatible with them being an irreducible specialization, as this merging into one composite node is only a proposition to simplify the lattice itself, and not something that should impact the underlying source code.

[2]http://fcaparser.googlecode.com/
[3]http://jmemorize.sourceforge.net/

The complete lattice of the class shows that the class implements several concerns: observers, logging, and application startup. The application startup concern is illustrated by the presence of main, run, and startStats methods. In the code, main calls run to start the application, which in turn calls startStats to collect program statistics during program execution. The bulk of the nodes in the center shows that these nodes implement some coherent functionality, which is revealed by their interconnections. The disjoint branches of the lattice propose to decompose the class to encapsulate each concern into a separate class.

On a single class with few members, the FCA technique and the patterns we propose may seem like an overkill, but one must understand that, in practice, such tools would be used for all the classes (thousands) of a system and a user would not have the possibility to analyze each one independently. The patterns would be a help to point out the classes that offer the best opportunities for design improvement.

B. Class Coupling

Class coupling explores the relationship amongst different classes, by the way each one uses the members of the other classes. Similarly to attribute uses (Section IV-A) for classes, concepts resulting from this context represent high level features in packages by identifying common access to other class members.

- O = Classes;
- A = Class members (attributes and methods);
- R = The class Uses a member of another class.

The example will use the classes of package org.jhotdraw.geom of JHotDraw. The resulting lattice is presented in Figure 3. We may find the following patterns:

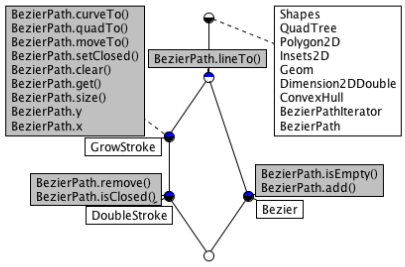

Fig. 3. Class Coupling lattice for `org.jhotdraw.geom` package in JHotDraw

- **Top black:** The pattern in this context shows the existence of nine classes in the package that do not access members of any of the classes in the package. In this specific case, the lack of communication is due to the fact that this package is an intermediary layer between java 2D graphics (java.awt.geom) and the rest of the JHotDraw framework.
- **Irreducible specialization:** There is a case of irreducible specialization that is similar to the one observed in Section IV-A.
- **Module:** There are two simple modules in this lattice. The first one consists of the lattice itself minus its top node. It gives us all the classes that interact together. The second one, two nodes on the left, for which we can draw the same conclusions that in example Section IV-A. Of course in such a small case, the simplifications that node aggregation would bring are not really needed.
- **Horizontal decomposition:** If we focus on the largest non-trivial module (whole lattice without the top black node), we can detect the presence of a horizontal decomposition pattern with two independent branches. They suggest two features: one that works with BezierPath s as containers (isEmpty(), add()) and the other one that sees them as graphical elements (moveTo(), isClosed(), ...). This shows that the class BezierPath implements two different concerns and requires refactoring different concerns in separate classes.

V. Prototype for Pattern Identification

We have provided concrete examples that show that the catalogue of patterns does reveal some important spots in concept lattices. We developed a tool in Moose [11] to automate the task of pattern identification in concept lattices. The tool supports the identification of all the patterns. We used the tool to extract the patterns from the lattice constructed from all classes in OpenBravo[4] using the following formal context:

- O =All classes;
- A = Method names;
- R = The class locally defines the method name

Table II provides a summary of the results produced by the tool. The resultant concept lattice is composed of 1053 nodes and 2260 connections. Of those nodes and connections, the user needs to study 154 nodes and 150 connections, if (s)he wants to explore all the patterns. This represents a reduction of 85% in the number of nodes to analyze and 93% in the number of connections.

TABLE II
Patterns in OpenBravo

# of formal objects	756
# of formal attributes	6658
# of nodes	1053
# of node connections	2260
Top	Top Black (1 node, 0 connections)
Bottom	-
Irreducible specializations	45 (90 nodes, 45 connections)
Horizontal decomposition	28 (28 nodes, 56 connections)
Modules	14 (49 nodes, 49 connections)
Total Nodes and connections in Patterns	154 nodes, 150 connections

The patterns in the lattice reveal: classes without methods or empty classes (Top black); hierarchies of classes having similar methods (Irreducible specialization); classes that do not have common methods with other classes (Horizontal decomposition). Modules in the lattice represent different methods that are common (duplicated) amongst the classes of

[4]http://www.openbravo.com/

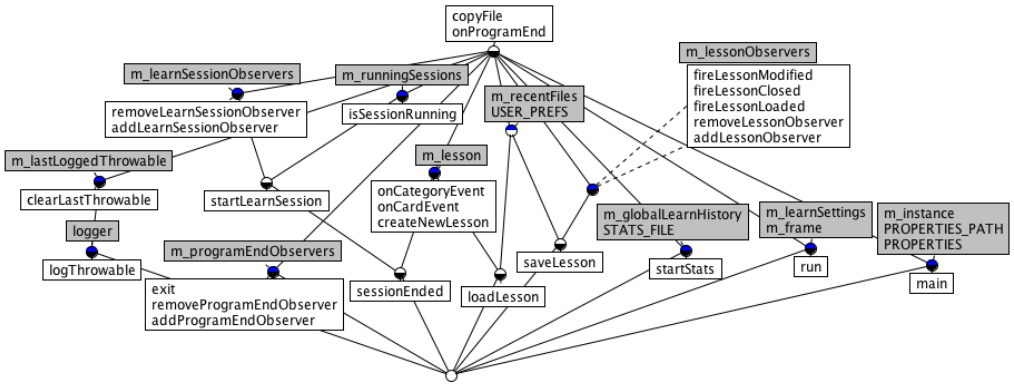

Fig. 2. Lattice for `Main` in JMemorize

the system. Hence, the modules nodes can be used to refactor these classes toward a better design.

VI. Conclusion

An inconvenience of using FCA is that lattices are complex and one needs to be knowledgeable about FCA to extract useful information. The paper presents patterns in concept lattices that can ease the task of lattice interpretation. We demonstrate that: The patterns reveal important information for the underlying context; The patterns identify pertinent results regardless of the context used for creating the formal concept; All mined patterns in the examples are pertinent, thus false-positive results are absent. A prototype is presented that searches for the patterns in lattices. In software reengineering, such a tool could support automated identification of problem areas. For our future work, we are interested in expanding the catalogue of patterns and mix our approach with filtering techniques.

References

[1] N. Anquetil. A comparison of graphs of concept for reverse engineering. IWPC, 2000.

[2] G. Arévalo, S. Ducasse, S. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Inf. Softw. Technol.*, 52:1167–1187, November 2010.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. Understanding classes using X-Ray views. In *MASPEGHI 2003 (ASE 2003)*, pages 9–18, Oct. 2003.

[4] T. Ball. The concept of dynamic analysis. In *ESEC/FSC'99*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[5] M. U. Bhatti, S. Ducasse, and M. Huchard. Reconsidering classes in procedural object-oriented code. In *WCRE*, 2008.

[6] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *Proceedings of IWPC '99*, pages 136–143. IEEE, May 1999.

[7] M. Dao, M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Improving generalization level in uml models iterative cross generalization in practice. In K. E. Wolff, H. D. Pfeiffer, and H. S. Delugach, editors, *ICCS*, volume 3127 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2004.

[8] U. Dekel and Y. Gil. Revealing class structure with concept lattices. In *WCRE*, pages 353–362. IEEE Press, Nov. 2003.

[9] A. Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of ICSE '99*, pages 246–255. ACM Press, 1999.

[10] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *OOPSLA*, pages 251–267, 1996.

[11] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, Sept. 2005. Tool demo.

[12] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[13] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization. In *ASE*, pages 431–434. IEEE, 2008.

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

[15] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices. In *Proceedings OOPSLA '93*, volume 28, pages 394–410, Oct. 1993.

[16] M. Habib, M. Huchard, and J. Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13(6):573–591, 1995.

[17] M. Huchard and H. Leblanc. Computing Interfaces in JAVA. In *Proceedings of ASE '2000*, pages 317–320, 2000.

[18] M. Huchard and H. Leblanc. Computing interfaces in java. In *ASE*, pages 317–320, 2000.

[19] P. Joshi and R. K. Joshi. Concept analysis for class cohesion. In *Proceedings CSMR 2009*, pages 237–240.

[20] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[21] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *ASE'05*, pages 66–75, Nov. 2005.

[22] C. Lindig. Mining patterns and violations using concept analysis. Technical report, Saarland University, Germany, 2007.

[23] K. Mens and T. Tourwé. Delving source code with formal concept analysis. *Comput. Lang. Syst. Struct.*, 31:183–197, October 2005.

[24] N. Moha, A. M. R. Hacene, P. Valtchev, and Y.-G. Guéhéneuc. Refactorings of design defects using relational concept analysis. ICFCA'08, pages 289–304. Springer-Verlag, 2008.

[25] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC '07*, pages 37–48, Washington, DC, USA, 2007. IEEE Computer Society.

[26] H. A. Sahraoui, H. Lounis, W. Melo, and H. Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering Journal*, 6(4):387–410, 1999.

[27] M. Siff and T. Reps. Identifying modules via concept analysis. *Transactions on Software Engineering*, 25(6):749–768, Nov. 1999.

[28] G. Snelting and F. Tip. Understanding Class Hierarchies Using Concept Analysis. *ACM TOPLAS*, pages 540–582, May 2000.

[29] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04*, pages 315–330, New York, NY, USA, 2004.

[30] A. Sutton and J. Maletic. Recovering uml class models from c++: A detailed explanation. *Inf. Softw. Technol.*, pages 212–229, March 2007.

[31] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In G. Stumme, editor, *Proceedings of ICFCA '03*. Springer-Verlag, Feb. 2003.

[32] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of ICSM '99*, pages 230–238. IEEE Computer Society Press, Oct. 1999.