Interactive Exploration of Structural Concepts in Code

Paul Heckmann and Daniel Speicher

Institute of Computer Science III, University of Bonn, Germany
heckmann@uni-bonn.de, dsp@cs.uni-bonn.de

Abstract. Understanding a software system is the first task in any reengineering activity. For this very challenging task one effective approach is to identify interesting and reoccuring structures in the software and to study these structures individually. In object-oriented software such structures typically consist of a few classes. The well known among them are called design pattern. Yet, which structures to look at in particular? Can we identify interesting structures that are not that well known? Which structures to be a clue to start with?
In this paper we extend a previously suggested approach of pattern mining using Formal Concept Analysis. We propose a way to eliminate redundant information in the overall analysis result. Besides that, we introduce two new features: The first feature is a *filtering element* that allows us to interactively and dynamically narrow the analysis space. The second is the *prominence* of a class - a measurement of the importance of the class to the overall system.
In an experimental evaluation we applied our approach on two software projects. In the first, JUnit, our tool guided the experimenter to central structures that can be found in the online documentation yet was unknown to the experimenter. In the second the tool led us to core structures of our own software.

Keywords: Design Pattern, Pattern Mining, Formal Concept Analysis, Interactive Software Exploration

1 Introduction

An object oriented software system essentially can be seen as a composition of structural concepts. Concepts in which classes and interfaces are connected with each other using building mechanisms like abstraction, inheritance, and composition to realize a certain functionality important to the respective part of the system. Some of these concepts reoccur over the entire project and constitute to a programs unique character, others reoccur yet only in certain parts suggesting core concepts of the program. Hence revealing any of these structural concepts can be a first important step to understanding the software itself, its character and its core functionality. But not all of these concepts arise by design, e.g. by using design patterns as introduced by [7]. They may arise implicitly and strongly depend on the developers style of solving a certain design problem.

In this work we propose an approach to mine structural concepts using a bi-clustering technique called Formal Concept Analysis (FCA) [8], building on a previous approach proposed by [13]. This technique allows us to group structures in source code into meaningful groups without requiring any knowledge on the to-analyze program nor the existence of a reference library of structures. We then improve this approach by the use of a more efficient mining algorithm and extend it by adding filtering features that, on one hand, allows us to interactively explore the structures in a program, and on the other hand supports us in finding those structural concepts constituting to its core functionality.

In section 2 we give a brief introduction to the very basic idea of FCA. In section 3 we reproduce the approach firstly introduced by [13] to apply FCA on our problem of mining structures in source code. In section 4 we present our extensions to this approach. Finally, in section 5 we validate the performance improvement on three software projects of different size and conduct two experiments to examine the practicability of our extensions.

2 Formal Concept Analysis

Formal Concept Analysis (FCA) [8][3] is a branch of lattice theory that allows us to identify meaningful groupings of *objects* G, i.e. quantities in a data set, that have common *attributes* M. In all the extent of this work, we are going to explain FCA on a very simple yet illustrative example in which we pick a set of birds as G and a set of bird characteristics as M. The triple (G, M, I) is called a *formal context*, where $I : G \times M \longrightarrow \{0, 1\}$ is an incidence function which returns 1 for a pair $(g, m) \in G \times M$ if g has attribute m, 0 otherwise. This formal context can be organized in an incidence matrix $\mathcal{M}$, as depicted in Table 1[1]. Here, $\mathcal{M}_{(i,j)}$ has an entry if object $I(i, j) = 1$.

Table 1. FCA bird example context.

	can fly	can swim	sings	migratory	monogamous
Ara	×				×
Bluejay	×		×		×
Kiwi					×
Mallard	×	×			
Pelican	×	×		×	

Using this context, FCA groups the objects and their attributes into *formal concepts*, listed in Table 2. Such a formal concept consists of two sets, an *extent* and an *intent*. The intent contains all *common* attributes that apply to the objects in the extent. In the same way all objects contained in the *extent* share all properties contained in the *intent*. Therefore a concept is a maximal collection of elements sharing common properties. Adding an attribute to a concept's intent

[1] It needs to be noted that ornithology usually is not part of our research. The data shown in Table 1 may not be entirely correct.

there would be at least one object in the extent that does not have this attribute. Adding an object to the extent there would be at least one attribute in the intent this object does not have. As a consequence the formal concepts build a *complete partial order* that can be written as a lattice. Table 2 in some way suggests this order by the increasing number objects and the decreasing number attributes from top to bottom.[2]

Table 2. Formal concepts for the context in Table 1.

i	Extent E_{c_i} ($\downarrow$)	Intent I_{c_i} ($\uparrow$)
c_1	{Pelican}	{can fly, can swim, migratory}
c_2	{Bluejay}	{can fly, sings, monogamous}
c_3	{Ara, Bluejay}	{can fly, monogamous}
c_4	{Mallard, Pelican}	{can fly, can swim}
c_5	{Kiwi, Ara, Bluejay}	{monogamous}
c_6	{Ara, Bluejay, Mallard, Pelican}	{can fly}

3 FCA Application

3.1 Setup of the Formal Context

We apply FCA on an object oriented software system by considering structures between classes and interfaces as the set G of FCA objects and class relationships that constitute to a structure as the set M of FCA attributes. A first logical step hence will be to define a structure model, i.e. the set of types of relationships we are looking for in a system. For this, we adapt relationships from LePUS3 [5], a modeling language for design patterns, and classify them into three structural and four behavioral relationship types, as listed in Table 3. All relationships are orthogonal to each other. For instance, the *calls* relationship between two classes A and B only applies if there is no forwards relationship between A and B.

Using this model to describe structures, we then adopt the FCA approach introduced by [13] and later improved by [1]. Here, having as input the set P of all classes in a software and a fixed order $n \in \mathbb{N}$ we gather those class substructures that contain n classes. Such a substructure can be described by two components, namely its n-tuple of classes in P^n (our FCA object) and a set of relationships between these classes (our FCA attribute). We illustrate this approach by the example structure in Figure 1 with $P = \{\text{A, B, C, D, E}\}$ and $n = 3$. The corresponding formal context to Figure 1 is shown in Table 4 while Table 5 lists the resulting formal concepts.

[2] In the context of this particular work, we are not making any use of this partial order yet.

Table 3. Our set of relationships used to describe structures in source code.

Relationship r	A class A is related to a class/interface B by r if
structural	
has	A has a one-to-one object association to B.
aggregates	A has a one-to-many object association to B.
specializes	A extends or implements B.
behavioral	
calls	a method in A calls a method in B.
forwards	a method in A calls a method in B that shares the same signature.
creates	a method in A calls the constructor of a class and binds the new instance to a variable of type B.
produces	a non-private method in A creates a new instance of B and returns it.

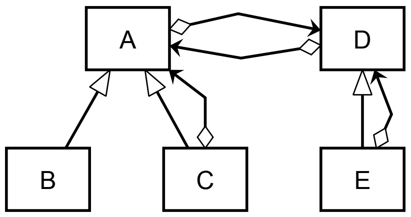

Fig. 1. A simple class structure that could be an input to our FCA approach. It contains five classes related to each other by two of our relationship types, *has* (simple association, diamond at source) and *specialization* (inheritance, triangle at target).

3.2 Iterative Concept Analysis

The problem of generating all concepts given a set its complexity is $\#P$-complete. More specifically, for a given context (G, M, I) the algorithm we used for our approach [10][11] has time complexity $O(|G|^2 \cdot |M| \cdot |C|)$, where C is the set of all concepts for the context. The space complexity is $O(|G| \cdot |M| \cdot |C|)$.
In order to reduce the set of concepts computed by FCA as well as its expected runtime, we apply FCA in two iterations. In the first iteration, only the three structural relationships in Table 3 are considered, creating concepts revealing *the definition of a system part.* In case we want to further examine *how such a definition works* with regard to its calling behavior, we apply FCA on it in a second iteration. This time we consider both, structural and behavioral relationships as attributes, however, vastly reduce the set of FCA objects as only the current concept's extent is used as input.[3]

[3] The idea was already proposed by [13], with the difference that they used as attribute augmentation the methods calling each other and their names.

Table 4. Formal context for the structure in Figure 1 and $n = 3$. Four connected substructures can be found. The attributes represent relationships and refer to the classes in the tuples by their indexes. For instance, the attribute *spec*(2, 1) can be read as "the element at index 2 specializes the element at index 1".

	spec(2,1)	*spec*(3,1)	*has*(3,1)	*has*(2,1)	*spec*(3,2)	*has*(3,2)	*has*(1,3)	*has*(1,2)
(A, B, C)	×	×	×					
(A, B, D)	×		×				×	
(A, C, D)	×		×	×			×	
(A, D, E)				×	×	×		×

Table 5. Formal concepts for the context in Table 4.

i	Extent E_{c_i} (↓)	Intent I_{c_i} (↑)
c_1	{(A, D, E)}	{*has*(1, 2), *has*(3, 2), *spec*(3, 2), *has*(2, 1)}
c_2	{(A, C, D)}	{*has*(3, 1), *has*(2, 1), *spec*(2, 1), *has*(1, 3)}
c_3	{(A, B, C)}	{*spec*(3, 1), *has*(3, 1), *spec*(2, 1)}
c_4	{(A, B, D), (A, C, D)}	{*has*(1, 3), *has*(3, 1), *spec*(2, 1)}
c_5	{(A, D, E), (A, C, D)}	{*has*(1, 2)}
c_6	{(A, B, D), (A, C, D), (A, B, C)}	{*spec*(2, 1), *has*(3, 1)}

3.3 Postprocessing

Removing Disconnected Structures Due to the way we construct the formal context basically two post-processing steps have to be taken. The first one is to remove concepts whose intent describe a disconnected graph. In Table 5 concept c_5 represents such a graph, as the nodes with the indexes 1 and 2 are connected with each other, the node with index 3 is not connected to any of the other two.

Merging Equivalent Structures For a context like ours FCA may produce concepts that are basically equivalent and can be merged. That is the case if there exists a permutation of the indexes of two concepts such that the intent of one concept can be mapped into the other. For instance, in Table 5 concept c_1 is equivalent to c_2 by the mapping $1 \rightarrow 3$, $2 \rightarrow 1$ and $3 \rightarrow 2$. To find such a mapping is a graph matching problem and hence not trivial. We used the VF2 algorithm proposed by [4] to accomplish this task, however, since the graphs we are trying to match are relatively small a naive depth-first search would work just as well.

Removing Redundant Information As an additional step we remove those concepts that contain redundant information. This is the case if the structure the concept describes contains a symmetric subgraph. A prominent example is the "star pattern" as depicted in Figure 2 (a). Here we can reduce the pattern to the one in Figure 2 (b) without losing any information. Since the reduced concepts represent structures of a lower order, we can ignore them.

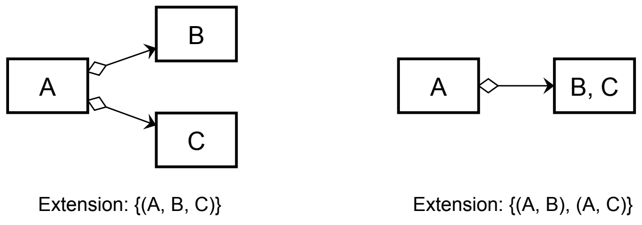

Fig. 2. A "star pattern" containing redundant information (left) and its reduction (right).

4 Filtering Features

4.1 Filter Elements

In order to dynamically change the space FCA is applied on we make use of *Filter Elements*. When gathering all substructures of given order in the setup of the formal context we proceed inductively, i.e. first compute all structures of order $n = 2$, then augment them to structures of order $n = 3$, etc. Before starting an analysis run we can declare classes as filter elements and aggregate them in a list. It is then guaranteed that in the first inductive step each structure of order $n = 2$ consists of at least one element from the filter element list. As a consequence, the structures serving as FCA objects then are the union of all structures that evolve around the classes in our list of filter elements.

4.2 Class Prominence

Taking a look at the extent of a formal concept we gain interesting information on single classes. One of such is the *prominence* of a class. For the extent $E_c \subset P^n$ of a formal concept c we can consider an index $r \in \{1, \ldots, n\}$ of the n-tuples in E_c as a *role* of concept c. A class $p \in P$ in any of the n-tuples in G_c at the index r then can be seen as a role player of role r. The set of all role players for a role r in a concept c we further refer to as $P_{c,r}$. In Table 5, concept c_6, we have three roles[4] according to their indexes in the 3-tuples, played by the following classes: $P_{c_6,1} = \{A\}$, $P_{c_6,2} = \{B, C\}$, $P_{c_6,3} = \{D, C\}$.

Given a class $p \in P$ and a role r in a concept c, we define by (3) its prominence $u(p, r, c)$ in r simply as the scaled frequency $\eta(p, r, c)$ (1) (2) of the class playing this role multiplied by the size of E_c.

$$\eta(p, r, c) := \frac{|\{e \in E_c \mid e_r = p\}|}{|E_c|} \tag{1}$$

[4] The number roles is determined by the order n.

$$\varphi(x) := x/(1-x) \tag{2}$$

$$u(p,r,c) := \varphi(\eta(p,r,c)) \cdot |E_c| \tag{3}$$

$$u(p) := \sum_{c,r} u(p,r,c) \Big/ \sum_{c,r,p} u(p,r,c) \tag{4}$$

In a final step we compute the absolute prominence $u(p)$ (4) of a class p simply by summing up the prominence values for a class over all roles over all concepts and normalize the outcome over the absolute prominence values of all classes.

It is worth noticing that $u(p)$ as defined in (3) has two factors. Previously proposed measures for interestingness are often composed of two values. One value measure the relevance of a finding the other the unexpectedness [6]. In our case the first factor emphasizes classes that are prevalent players of a role (of which only a few exist and therefore are unexpected). The second factor simply measure the size of the extent of the concept and thus its relevance.
Throughout the concepts listed in Table 5, apparently the class A is more prominent than any of the other four classes. Pretending that we replace class A with another class X in concepts c_1-c_3, class A would still gain the highest prominence value as it is the only player of role $r = 1$ in concepts c_4-c_6, which all have a larger extent.

The prominence of a class can serve us in two ways: First, the more prominent a class the higher the probability that it plays a role in a core concept of a software project. One can see the prominence as a gravitation of a class. The higher, the greater is the part of a software project that is 'attached' to this class. Second, it could help us determining appropriate filter elements. The lower the prominence of a class, the smaller the expected number computed concepts.

5 Case Studies

5.1 Our Tool

We implemented our approach as part of the *Cultivate*[5] plugin for the *Eclipse*[6] IDE. Cultivate is a code analysis tool for Java programs. It in turn bases on the *JTransformer*[7] plugin which provides a Prolog factbase that represents the full abstract syntax tree of the to-analyze Java program. Cultivate implements several program analyses written in Prolog that are applied on this factbase.
To compute the formal concepts we use a relatively young algorithm proposed by [10]. This algorithm is particularly suitable to our approach compared to the algorithm [8] used by previous works because it saves time not carrying the hierarchical order of the concepts required to build the concept lattice. Due to

[5] http://sewiki.iai.uni-bonn.de/research/cultivate/start
[6] http://www.eclipse.org/
[7] http://sewiki.iai.uni-bonn.de/research/jtransformer/start

our post-processing reorganizing the computed concepts this additional information is useless to us anyway. Secondly, this algorithm allows us to parallel the computation, distributing the computational load over several CPUs.

5.2 Data Set

We apply our tool on three different Java projects:

- *JUnit*[8] 4.7, which is a testing framework for Java code of smaller size.
- *Cultivate*, the framework our tool is based on. It consists of a platform providing utility classes and the engine on one hand and several addons on the other hand that build upon this platform but not upon each other. As a consequence the overall cohesion in this project is very low. Also we are familiar with its domain, which makes it easier to evaluate our findings.
- *JHotDraw*[9] 7, a free Java-based framework for creating graphical editors. In contrast to Cultivate we are not familiar with this project, yet enjoy it to be well documented. Also the overall cohesion of the project compared to Cultivate is fairly high.

For the sake of simplicity we consider only the set of core classes (ignoring external library classes) that fulfill the following requirements: The class neither is of a basic type (*Integer*, *Double*, ...), an enumeration type nor an anonymous class.

5.3 Performance

We ran our analyses on an Intel Quad Core @2.83 GHz with 4 GB RAM under normal load. Table 6 shows the runtime behavior we observed and number concepts computed for each of the three sample projects and with regard to the order n. For $n \geq 5$ on Cultivate our tool failed due to lack of memory. This

Table 6. Observed runtime behavior of our tool in seconds and number concepts computed in the first iteration.

	JUnit 4.7		Cultivate		JHotDraw 7	
#classes	143		607		625	
	runtime	concepts	runtime	concepts	runtime	concepts
$n = 2$	<1s	5	∼2s	7	∼3s	9
$n = 3$	<1s	16	∼3s	45	∼14s	38
$n = 4$	∼1s	24	187s	154	130s	141

may or may not be caused by Prolog and the fact it loads the entire factbase into the main memory as well as it caches its query results. The reason why

[8] http://www.junit.org

[9] http://sourceforge.net/projects/jhotdraw

the analysis on JHotDraw is faster than on Cultivate for $n > 3$ we ascribe to Cultivate being way less cohesive than JHotDraw, which eventually leads to a faster growth of concepts in n. Yet the actual number concepts in both projects are insignificantly different what suggests a fairly large impact of our additional post-processing.
Recalling previous observations made by [1], using the Ganter algorithm [8] the analysis of a sample project written in Smalltalk with 167 classes and $n = 4$ took approximately two days.[10] Compared to this, our own results by far excel our expectations and prove this technique to be a time-efficient way to analyze software projects even of larger scale.

6 Example Applications

6.1 Experiment 1: JUnit

In a first experiment we pick the smaller of our sample projects, JUnit, and let one of the authors apply our tool on it with the goal to yield most relevant structures of the project in at most five analysis steps. The project is well-documented and makes extended use of design patterns, however, the experimenter neither is familiar with the project nor with its documentation at the time of execution.
As a first step, the experimenter runs an analysis on structures of order $n = 2$. Despite our expectations we found $n = 2$ particularly instructive, as its corresponding concepts are small in number, easy to understand and most often already reveal those atomic relationships between two classes larger patterns are only based on. The analysis computes five concepts of which the one depicted in Figure 3 (a) catches the experimenters attention as it is one of two with more than one relationship. This pattern suggests the implementation of a tree structure using the *Composite* pattern [7]. In order to check this presumption the experimenter runs an analysis on order $n = 3$ using the class `TestSuite` as a filter element. Since this class has a significantly smaller prominence than `Test`, it is more suitable as a filter. Two concepts are computed, one of them actually representing the Composite pattern as depicted in Figure 3 (b).
As a specialization of `Test` we find a class called `TestDecorator` in Figure 3 (b) which suggests the implementation of the *Decorator* pattern [7]. Following the same procedure as before (using `TestDecorator` as filter element) we can verify our presumption.
Rechecking our findings so far with the JUnit documentation we can verify the composite pattern instance as one most relevant to the base framework, while the decorator pattern instance is particularly important to the extensions part of Junit which can be used by developers to implement and plug-in custom test definitions.

[10] A less advanced hardware may have an impact on these stats, too, considering the previous observations date back eight years.

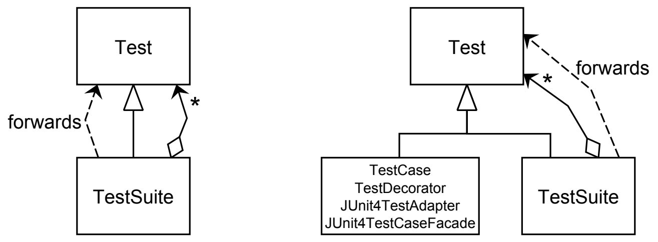

Fig. 3. Composite pattern candidate (left) and verified instance (right) in JUnit 4.7.

6.2 Experiment 2: Cultivate

In a second experiment we wanted to examine the precision of the prominence calculation and how it can be exploited to find core concepts in a software project. For this, we use our own project, Cultivate. Firstly, because we are familiar with it and can assess the validity of the computed prominence values. Secondly, Cultivate basically is a platform with a few core classes that, however, are extensively used by the addons that build on the platform.
We run an analysis on structures of order $n = 3$ and retrieve the list of all occurring classes ranked by their prominence. The one with the highest prominence value is `CultivateViewPart` (∼12%), the next-prominent class is `BaseQuery` (∼3%). In both cases we agree with the tool: `CultivateViewPart` is a class used as an abstract view part that follows selections in the workbench and manages the subscription of analyses on the corresponding software projects. It is basically inherited from all add-on projects that provide a workbench view and in fact is one of the central classes in Cultivate. `BaseQuery` is *the* abstract class to query analyses on Prolog side, specialized by 63 different classes inside the add-on parts.
We declare `CultivateViewPart` as filter element and run the analysis again. The result is a set of 14 concepts of which two describe exactly the main responsibilities of this class, depicted in Figure 4.

7 Discussion

Judging from the observed runtime performance, we can see that FCA is a practical technique for mining structures in software projects even of larger size. We find a promising approach to assess the relevance and importance of certain classes of the software project which may lead us to those parts of the software that constitute its most important functionality. Yet we conclude that this assessment still requires some fine-tuning. Having such an assessment we can exploit it either as a clue to search and identify core concepts of the corresponding project or as an assistance in choosing appropriate filter elements to narrow down the space the analysis is applied on.

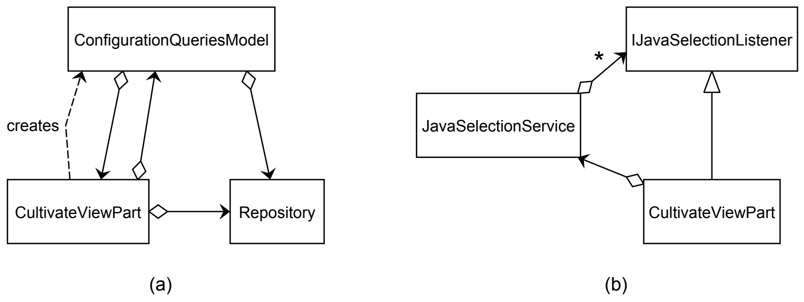

Fig. 4. Two core concepts of the Cultivate project. In Figure (a) the `CultivateViewPart` creates and attaches to a `ConfigurationQueriesModel` object which then retrieves a `Repository` for the currently selected project and handles query subscriptions on that repository. (b) describes an *Observer* pattern [7], in which `CultivateViewPart` is an observer, `JavaSelectionService` handles workbench selections.

In this work, we did not yet make use of the actual strength of FCA: That is, to create a *reverse partial order* between intent and extent of all concepts[11]. This feature may allow us to introduce two further measurements: Abstractness, resp. specificity of a software structure. Apart from that, it seems reasonable to apply our idea of class and structure measurement using clustering techniques with less high complexity [14][9][2] and may be addressed in future work.

8 Related Work

Formal Concept Analysis (FCA) was firstly proposed by [8] as a branch of lattice theory. The first effort towards structure mining in source code using FCA was achieved by [13]. Their approach then was later refined by [1] who reduced the number FCA objects and hence improved the overall runtime. Further structure mining efforts for object-oriented systems have been achieved by [14][9][2] who used subgraph matching to group same structures formed by classes. We adopted the approach by [13] refined by [1], enhanced the set of FCA attributes, i.e. class relationships, using a set of relationships based on the modeling language LePUS3 by [5] and exchanged the previously used algorithm to compute formal concepts by a relatively young algorithm that was proposed by [10]. A slight connection to our findings of the importance of order 2 structures can be drawn to [12], who tried to decompose design patterns into their elemental parts.

[11] As you can see in Table 2, the number attributes of all concepts stand in an inverse relation to the number objects.

References

1. G. Arévalo. *High Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Bern, 2004. 262, 268, 270
2. A. Belderrar, S. Kpodjedo, Y.-G. Guéhéneuc, G. Antoniol, and P. Galinier. Subgraph Mining: Identifying Micro-architextures in Evolving Object-oriented Software. *Europ. Conference on Software Maintenance and Reengineering*, 2011. 270
3. C. Carpineto and G. Romano. *Concept Data Analysis, Theory and Applications*. Wiley & Sons, 2004. 261
4. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001. 264
5. A. H. Eden, Y. Hirshfeld, and A. Yehudai. LePUS - A Declarative Pattern Specification Language. Technical report, 1998. 262, 270
6. I. Exman, G. Amar, and R. Shaltiel. The Interestingness Tool for Search in the Web. In *Proceedings of the Third International Workshop on Software Knowledge (SKY 2012)*, Barcelona, Spain, 2012. 266
7. E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1994. 260, 268, 270
8. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1998. 260, 261, 266, 268, 270
9. M. Gupta and A. Pande. Design patterns mining using subgraph isomorphism: Relational view. *International Journal of Software Engineering and Its Applications (IJSEIA)*, 2011. 270
10. P. Krajca, J. Outrata, and V. Vychodil. Parallel Recursive Algorithm for FCA. In *Concept Lattices and Their Applications (CLA)*, pages 71–82, 2008. 263, 266, 270
11. S. O. Kuznetsov. Learning of Simple Conceptual Graphs from Positive and Negative Examples. *Principles of Data Mining and Knowledge Discovery*, 1999. 263
12. J. M. Smith and D. Stotts. Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture. In *IEEE/NASA Software Engineering Workshop*, pages 183–190, 2002. 270
13. P. Tonella and G. Antoniol. Object Oriented Design Pattern Inference. In *Proceedings of ICSM*, page 230ff. IEEE Computer Society Press, 1999. 261, 262, 263, 270
14. Z.-X. Zhang, Q.-H. Li, and Ke-Rongben. A New Method for Design Pattern Mining. *International Conference on Machine Learning and Cybernetics*, 2004. 270