Detecting Software Patterns using Formal Concept Analysis

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Frank Buchli

September 2003

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Gabriela Arévalo

Institut für Informatik und angewandte Mathematik

Further information about this work, the used tools and an *online* version of this document can be found at:
`http://www.iam.unibe.ch/~scg/`

The address of the author:

Frank Buchli
Benedikt Hugistrasse 13
CH-4500 Solothurn
`fr@nk.buchli.org`
`http://www.buchli.org/frank`

or

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
`http://www.iam.unibe.ch/~scg/`

Abstract

Redocumentation and design recovery are two important areas of reverse engineering. Detection of recurring organizations of classes and communicating objects, called Software Patterns, supports this process. Many approaches to detect Software Patterns have been published in the past years.

Most of these approaches need a pattern library as reference. Personal coding style and domain specific requirements lead to creating new patterns or adapting existing ones and make those approaches fail. The second problem is that the found patterns of those methods are presented without connection to the other patterns. To gain an overview of the whole system and its mechanisms, we propose to set the patterns in relation each other.

Our work shows a method to detect Software Patterns using Formal Concept Analysis (FCA). The advantage of this approach is that no reference library is needed and the results are set in relation each other.

FCA is a mathematical theory which detects the presence of groups of classes which instantiate a common, repeated pattern. Those found patterns are presented in a lattice, a partial order relation among the patterns, which allows us to explore the pattern which are in relation to them.

We implemented a prototype tool *ConAn PaDi* which navigates with the *Fish Eye View* technique over the patterns. For validation we applied this tool to three mid-sized Smalltalk applications.

Acknowledgments

It was a great opportunity what the Software Composition Group of Prof. Dr. Oscar Nierstrasz was offering me: I could write a master thesis in the field I was most interested in: software engineering. It was extremely interesting for me to live in this world of academic research. I got introduced in the world of papers, citations, conferences, *etc*.

I want to thank the people of this group. First of all my supervisor Gabriela Arévalo, she helped me whenever I needed her assistance. It was a pleasure for me to develop with her the *ConAn* framework. Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Michele Lanza took their time and helped me with their great knowledge, thanks a lot.

Thanks to all the students from the pool for not let me suffering alone. Special thanks to Tobias Aebi and Daniele Talerico for sharing so many ideas, problems and solutions.

Thanks goes as well to my family which made all this possible. Without their continuous support and advise I would never be where I am now.

Last but not least, I want to thank Tinka, my love. Thank for being patient and accepting my time consuming commitment for AIESEC, I had beside my studies.

Finally, I want to share to the reader one statement that came in this work true for me once more:

The more you give, the more you get.

Frank Buchli,
September 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Patterns in Reverse Engineering

The major issue of legacy systems nowadays is maintenance. Systems developed during a long period become hard to maintain and adapt. Requirements change, platforms change and if a system is not properly maintained, its usefulness decays over time [LEHM 85]. A cost saving solution is maintenance and reengineering. The first step of the reengineering process is reverse engineering. Reverse engineering helps to clarify the structure by extracting information and providing high-level views on the subject system.

Among these high level views, Software Patterns are seen as a logical basis for design recovery and redocumentation, which are two major areas in reverse engineering [CHIK 90]. The implicit design may be recovered by recognizing occurrences of known Software Patterns in source code [NIER 02]. Due to the fact that in the literature several definitions of Software Patterns can be found [APPL] according to the context they are applied in, we clarify what we mean in our work by *Software Patterns*. We consider a Software Pattern structural aspects of objects and classes that are characterized and connected by different kinds of structural relationships. With this definition we want to distinguish clearly from the well known Design Patterns [GAMM 95]: Software patterns do not necessarily have as intention to solve a general design problem.

1.2 Formal Concept Analysis

Formal Concept Analysis (FCA) is based on mathematical order theory and is a branch of lattice theory [GANT 99]. FCA groups *elements* that have common *properties*. Those groups are called *concepts*. A concept is a maximal collection of elements that exhibit common properties, *i.e.*, a grouping of all the elements that share a set of properties.

1.3 Contributions

This work presents a methodology how to use Formal Concept Analysis to detect Software Patterns. We show how the detection of Software Patterns is useful in software engineering. The strength, such as finding relationships between the patterns, and the weakness, such as scalability, of our approach are reported. Last but not least, a tool which implements our approach is introduced. The work is based on the ideas reported in [TONE 99]. The additional contribution of our work is:

- We define the pattern neighborhood; this means that we set the patterns in relation each other. For example, it is possible to detect patterns which are almost like another pattern.
- We propose a improvement for the inductive algorithm of [TONE 99] which makes the algorithm faster. The idea is to reduce the amount of elements once more. Reducing the elements makes the a shorter calculation time for the formal concept generation.
- We propose to take the information from a language independent meta-model instead from the source code itself. This makes the approach more powerful because it can be applied on applications in different programming languages.
- We propose to clear separate the calculation of the patterns from the analysis process. This allows to let run once the time consuming calculation of the patterns and store them in a repository. To analyze, the user can apply different filters on this repository and play like this with the results.
- To get a faster overview, we propose that in the post process the found patterns are compared with a reference library of well known (design) patterns. The so found patterns become the name of the reference.
- All these ideas are implemented in the tool *ConAn PaDi*. The tool is able to navigate through the found patterns by using the *Fish Eye View* technique.

1.4 Organization of the Document

This document is structured as follows:

- In *Chapter 2* we introduce the problems of object-oriented reverse engineering. Software Patterns are explained and an example is given. In the second part, we set our scope: Detecting Software Patterns using Formal Concept Analysis.
- *Chapter 3* introduces the mathematical theory of FCA and how FCA can be used in software engineering in general.

- *Chapter 4* describes our methodology. The prerequisites for FCA are defined and the post processes on the FCA concepts. The term *pattern neighborhood* is defined.
- *Chapter 5* validates our approach on three mid-size Smalltalk applications. The setup for the case study is explained.
- In *Chapter 6* the tool, *ConAn PaDi* is introduced. Its components *Moose*, a meta-model framework and *ConAn*, a framework for FCA, are presented.
- *Chapter 7* gives an overview of other publications about pattern detection.
- In *Chapter 8* we conclude the main contributions of our work and give an outlook on possible future work in the research field of detecting Software Patterns with FCA.
- *Appendix A* summarizes all the explored formal properties.

Chapter 2

Problems with Detecting Software Patterns

2.1 Introduction

This chapter firstly introduces terminology used in the context where we applied our approach: object-oriented reverse engineering. It then introduces the main topic of this work – software patterns – and shows how they are beneficial for reverse engineering. Finally it gives an overview of the scope and goals of our approach.

2.2 Object-Oriented Reverse Engineering

Maintenance, reengineering, and evolution of software systems has become a vital matter in today's software industry. The law of *software entropy* dictates that most systems tend to gradually decay in quality over time, unless the system is maintained and adapted to the evolving requirements [LANZ 03]. Lehman's software evolution laws state, that software systems must be continually adapted, otherwise they become progressively less satisfactory [LEHM 85]. For example, the customer needs support for new platforms, to embrace emerging standards or better security features. On the other hand, a complete redesign may be impractical or too expensive. In such cases it is more advisable to maintain, reengineer, and evolve such systems by adapting them to new requirements [CASA 98][RUGA 98] in order to extend their lifetime and to increase the return of investment of their owners. The lifetime of software system can be extended by maintaining and/or reengineering it.

In order to understand in which software engineering phases we are able to apply our approach, we define the common terminology used in this context:

Reengineering ...is the examination and alteration of a subject system to reconstitute it in a new form... It generally includes some form of reverse engineering (to achieve a more

abstract definition) followed by some form of forward engineering or restructuring. [CHIK 90]

Before a software system can be reengineered the system must be *reverse engineered, e.g.,* a mental model of the software needs to be built which allows for taking informed decisions [LANZ 03].

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. [CHIK 90]

Reverse engineering itself is part of the reengineering process as illustrated in Figure 2.1. Reengineering depends on reverse engineering because in order to modify that system needs to be understood first. Changing a large and complex system without enough knowledge of its inner structure, will almost certainly trigger unwanted side effects which could make the system inoperable [LANZ 99].

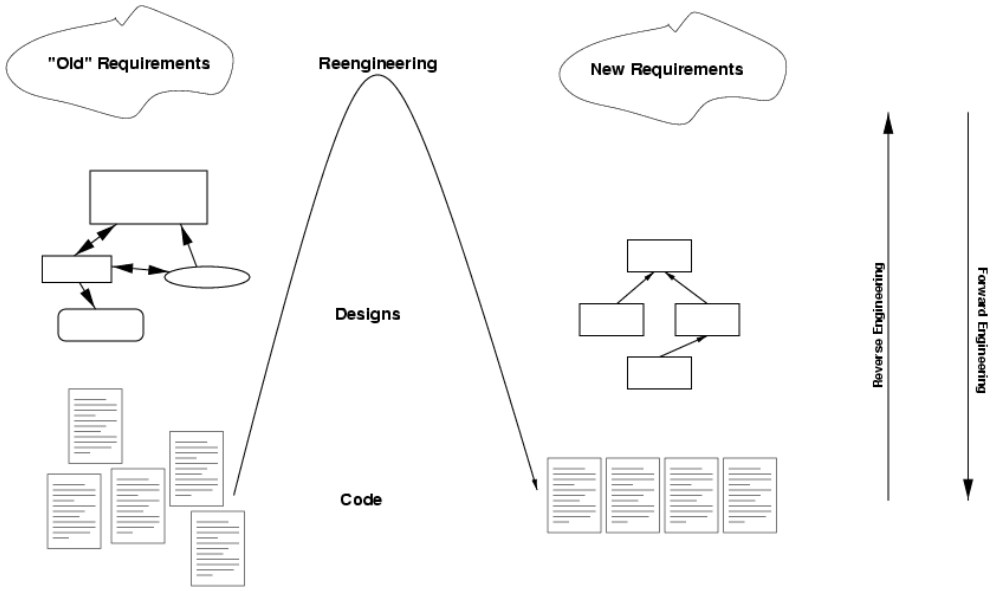


Figure 2.1: The reengineering life-cycle of software

Chikofsky and Cross state that *The primary purpose of **reverse engineering** a software system is to increase the overall comprehensibility of the system for both maintenance and new development* [CHIK 90]. They list six key reverse engineering objectives:

1. Cope with complexity
2. Generate alternate views
3. Recover lost information
4. Detect side effects

5. Synthesize higher abstractions
6. Facilitate reuse

Two important areas, that are widely referred to, are redocumentation and design recovery. [CHIK 90].

Redocumentation ... is the creation or revision of a semantically equivalent representation within the same relative abstraction level. [CHIK 90].

Redocumentation is the simplest and oldest form of reverse engineering [CHIK 90]. Typical tools for redocumentation are diagram generators and cross-reference listing generators. A key goal of these tools is to provide easier ways to visualize relationships among program components.

Design Recovery ... is a subset of reverse engineering in which domain knowledge, external information and deduction or fuzzy reasoning are added to the observation of the subject system to identify meaningful higher level abstractions.... [CHIK 90]

Design recovery is a key aspect because we get the programmer's original intention with the recovered design. In the best case design recovery reproduces all the information required for a person to fully understand what a program does.

Lanza points out that object-oriented systems pose many additional challenges in comparison with procedural systems [LANZ 03]:

- Polymorphism and late-binding make traditional tool analyzers, like program slicers, inadequate. Data-flow analyzers are more complex to build especially for dynamically typed languages.
- The use of inheritance and incremental class definitions, together with polymorphism and overriding, make applications more difficult to understand.
- The domain model of the applications is spread over classes residing in different hierarchies and/or subsystems and it can be difficult to pinpoint the location of a certain functionality.
- In procedural systems it is obvious where the starting point is and a top-down reverse engineering approaches can work. In the case of object-oriented systems the first question a reverse engineer has to answer is where to start the reverse engineering process. Moreover the program execution flow is not trivial and has to be found out.

2.3 Design Patterns

Design Patterns are beneficial to the forward engineering process [SCHM 95][BECK 94], but can also represent relevant information about the system in the reverse engineering phase [TONE 99].

Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [GAMM 95]

(Design) patterns are recurring organizations of classes and communicating objects that can be found in many object-oriented systems, and are employed to make the design more flexible, elegant and reusable. A very popular object oriented design patterns collection is listed in [GAMM 95]. Gamma et al. introduce 23 design patterns, as well as their implementation in Smalltalk and C++. From a program understanding and maintenance perspective a pattern provides knowledge about the role of each class in the pattern organization. This information can be used to find out the role of the class in the whole system as well.

Gamma et al. classify the Design Patterns into three sections, according to their purpose: *creational*, *structural* and *behavioral* ones. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility [GAMM 95].

Example: Composite Pattern

An often referred and used structural Design Pattern is the Composite Pattern, which we use as an example here. The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Graph applications, like drawing editors and schematic capture systems, let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical, primitives such as Text and Lines.

But there is a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. The application becomes more complex distinguishing these objects. The composite pattern describes how to use recursive composition so that users do not have to make this distinction. The structure of this example is shown in Figure 2.2.

This pattern is described more in detail in [GAMM 95].

2.4 Software Patterns

In this work we use the term *Software Pattern* to distinguish clearly from *Design Pattern*.

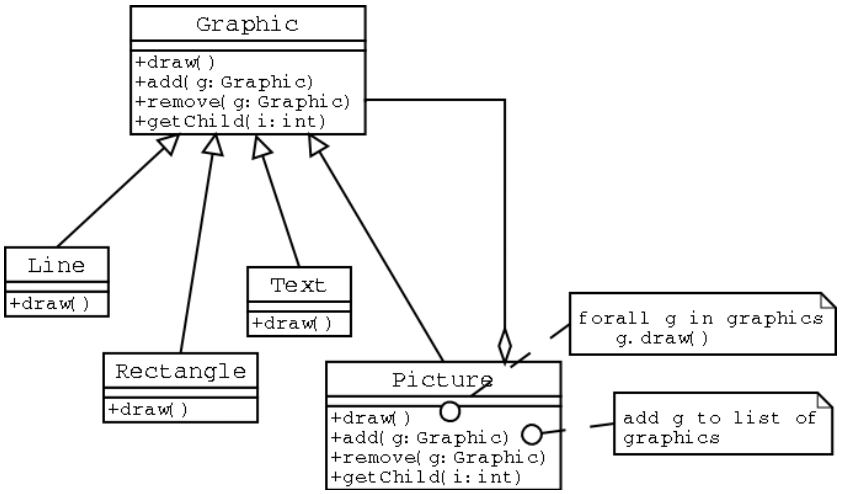


Figure 2.2: Example of a Composite Pattern

Software patterns are structural aspects of a set of objects and classes that are characterized and connected by different kinds of structural relationships.

This definition is very general and allows different structures to be called Software Pattern. But this is exactly what we are looking for: Identifying set of structures that can be concerned as Software Pattern. We want to be able to find *any* pattern in the system.

2.5 Scope: Detecting Software Patterns

The detection of patterns supports redocumentation by generating alternate views, recovering lost information and synthesizing higher abstractions. Having detected patterns makes modifications easier [BECK 94].

We want to find out how useful this approach is to reach the previously listed objectives of reverse engineering. This six objectives listed are too general to be directly usable as concrete goals. Therefore we settle down the following concrete goals for our approach:

1. Gain an overview of the system structure
2. Locate and understand relations such as inheritance, accesses and invocations.
3. Detect class dependencies
4. Understand the class roles
5. Identify the possible presence of classical Design Patterns
6. Identify the neighborhood of a pattern (related patterns, *i.e.*, a pattern included by another)
7. Find candidate classes for restructuring
8. Identify coding styles

Before analyzing the patterns, they have to be found. There are several techniques to infer software patterns from a software systems, *e.g.* subgraph matching techniques with a reference library [SEEM 98] [KELL 99][SCHA 98][NIER 02] or predefined Prolog rules [KRAM 96]. We have chosen *Formal Concept Analysis*, a mathematical theory which allows automatically generate grouping of elements. The advantages of FCA are:

- FCA can infer any kind of pattern in the system. It does not need a predefined library with the pattern definition, therefore we do not have to make any assumption on the existence of a pattern. The technique can detect unpredictable relationships *Note:* In order to assigning in a postprocess common used names to the found patterns, such a library is nevertheless needed. But this is just to attach the results to well known predefined patterns. Having some of the patterns identified as well known pattern the user gains a sooner an overview of the results.
- We can represent design constraints by *simple* properties. Simple means that properties are simple to read, do not need any previous calculation and can be obtained directly from the source code (or its representative meta-model). (*E.g.:* inheritance, abstract classes, defined methods, attributes of classes)
- We have a patrial order on the concepts and this can be analyzed to have some information about relationships between the design fragments in a system, what we call *pattern neighborhood*.

The next chapter introduces how FCA can be used in general in software engineering. Besides finding out how adequate the approach is for software engineering, we have a look at the approach itself and see how we can improve it. Concrete ideas are listed below:

- Are the properties (*e.g.* isSubclass, accesses) presented in [KRAM 96],[SEEM 98] and [TONE 99] useful to detect structural (un)known patterns in an application? Are more generic or more specific ones needed?
- Can we complement the structural properties with behavioral ones to have more complete information? Could we explore behavioral patterns?
- Do we find heuristics to make the algorithm faster?
- How can we evaluate the information of the lattice?

Chapter 3

Software Engineering with FCA

3.1 Introduction

This chapter firstly provides an introduction to the mathematical field of FCA. The second section gives an overview about where and how FCA is already applied in software engineering.

3.2 Formal Concept Analysis

Formal Concept Analysis (FCA) [GANT 99] (also known as Galois lattices [WILL 81]) is a branch of lattice theory that allows us to identify meaningful groupings of *elements* (referred to as *objects* in FCA literature) that have common *properties* (referred to as *attributes* in FCA literature) [1].

One illustrative example about a crude classification of a group of mammals (*Cats, Gibbons, Dolphins, Humans,* and *Whales*) will lead us through the theoretical explanations. We consider five possible characteristics: *four-legged, hair-covered, intelligent, marine,* and *thumbed*. Table 3.1 shows the relationships between the mammals and its characteristics.

But first of all, we need to understand a few definitions to see how we analyze the information provided by FCA.

3.2.1 Context and Concepts

The initial starting point in using FCA is setting up a *context*. A **context** is a triple:

$$\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I}).$$

[1] We prefer to use the terms *element* and *property* instead of *object* and *attribute* because the latter terms have a specific meaning in the object-oriented paradigm.

		$\mathcal{P}$				
	$\mathcal{I}$	four-legged	hair-covered	intelligent	marine	thumbed
$\mathcal{E}$	Cats	×	×			
	Dogs	×	×			
	Dolphins			×	×	
	Gibbons		×	×		×
	Humans			×		×
	Whales			×	×	

Table 3.1: Mammal example: Table $\mathcal{T}$ represents the binary relations

$\mathcal{E}$ is a finite set of *elements*, $\mathcal{P}$ is a finite set of *properties* and $\mathcal{I}$ is a binary relation between $\mathcal{E}$ and $\mathcal{P}$: $\mathcal{I} \subseteq \mathcal{E} \times \mathcal{P}$. $\mathcal{I}$ is usually represented as a table $\mathcal{T}$. The binary relation in our example is shown in Table 3.1, where we see that our *elements* are the animals and *properties* are its characteristics. Then we see that the tuple *(Whales, marine)* is in $\mathcal{I}$ but *(Cats, intelligent)* is not.

Let $X \subseteq \mathcal{E}$ and $Y \subseteq \mathcal{P}$. The mappings:

$$\sigma(X) = \{p \in \mathcal{P} \mid \forall e \in X : (p, e) \in \mathcal{I}\},$$

the *common properties* of X, and

$$\tau(Y) = \{e \in \mathcal{E} \mid \forall p \in Y : (p, e) \in \mathcal{I}\},$$

the *common elements* of Y, form a *Galois connection*. That is, the mappings are *antimonotone*:

$$X_1 \subseteq X_2 \rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$
$$Y_1 \subseteq Y_2 \rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and *extensive*:

$$X \subseteq \tau(\sigma(X)) \quad \text{and} \quad Y \subseteq \sigma(\tau(Y)).$$

In the mammal example:
$\sigma(\{Cats, Gibbons\}) = \{hair\text{-}covered\}$
$\tau(\{marine\}) = \{Dolphins, Whales\}$

Based on the previous definitions, we define the term concept. A **concept** is a pair of sets: a set of elements (the *extent*) and a set of properties (the *intent*) (X, Y) such that [2]:

$$Y = \sigma(X) \quad \text{and} \quad X = \tau(Y).$$

[2]The notation in [GODI 98] and [GANT 99] is different to denote the σ or the τ. They are defined in the following way. Given a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$, and two sets $X \subseteq \mathcal{E}$ and $Y \subseteq \mathcal{P}$:

$$X' = \{p \in \mathcal{P} | \forall e \in X : (p, e) \in \mathcal{I}\}$$
$$Y' = \{e \in \mathcal{E} | \forall p \in Y : (p, e) \in \mathcal{I}\}$$

Thus using this notation, [GANT 99] summarizes the naive possibilities of generating all concepts as: *Each*

Therefore a concept is a maximal collection of elements sharing common properties. Informally, such a concept corresponds to a maximal rectangle in the cross-table $\mathcal{T}$: any $e \in \mathcal{E}$ has all properties in $\mathcal{P}$, and all properties $p \in \mathcal{P}$ fit to all elements in $\mathcal{E}$. In the mammal example, $(\{Cats, Dogs\}, \{four\text{-}legged, hair\text{-}covered\})$ is a concept, whereas $(\{Cats, Gibbons\}, \{hair\text{-}covered\})$ is not a concept. Although $\sigma(\{Cats, Gibbons\}) = \{hair\text{-}covered\})$, $\tau(\{hair\text{-}covered\}) = \{Cats, Dogs, Gibbons\}$ shows that it is not a concept. Figure 3.2 shows the complete list of concepts. It is important to note that concepts are invariant against row or column permutations in the cross-table $\mathcal{T}$.

top	({ Cats, Gibbons, Dogs, Dolphins, Humans, Whales }, $\emptyset$)
c_6	({ Gibbons, Dolphins, Humans, Whales } , { intelligent })
c_5	({ Cats, Gibbons, Dogs } , { hair-covered })
c_4	({ Dolphins, Whales } , { intelligent, marine })
c_3	({ Gibbons, Humans } , { intelligent, thumbed })
c_2	({ Cats, Dogs } , { hair-covered, four-legged })
c_1	({ Gibbons } , { hair-covered, intelligent, thumbed })
bottom	($\emptyset$, { four-legged, hair-covered, intelligent, marine, thumbed })

Table 3.2: Concepts of the mammal example

3.2.2 Concept Lattice

The set of all the concepts of a given context forms a *complete partial order*. Thus we define that a concept (X_0, Y_0) is a **subconcept** of concept (X_1, Y_1), denoted by

$$(X_0, Y_0) \sqsubseteq (X_1, Y_1)$$

if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$).

For instance, $(\{Dolphin, Whales\}, \{intelligent, marine\})$ is a subconcept of $(\{Gibbons, Dolphins, Humans, Whales\}, \{intelligent\})$. Thus the set of concept constitutes a *concept lattice* $\mathcal{L}(\mathcal{T})$ [BIRK 40]. The concept lattice for the mammal example is shown in Figure 3.1

Each node in the lattice represents a concept and they are shown in Table 3.2. Given two elements (E_1, P_1) and (E_2, P_2) in the concept lattice, their *infimum* or *meet* is defined as:

$$(E_1, P_2) \sqcap (E_2, P_2) = (E_1 \cap E_2, \sigma(E_1 \cap E_2)),$$

and their *supremum* or *join* as

$$(E_1, P_2) \sqcup (E_2, P_2) = (\tau(P_1 \cap P_2), P_1 \cap P_2),$$

concept of a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$ has the form (X'', X') for some subset $X \subseteq \mathcal{E}$ and the form (Y', Y'') for some subset $Y \subseteq \mathcal{P}$. Conversely, all such pairs are concepts. Every extent is the intersection of property extents and every intent is the intersection of element intents. We use in this work the notation with σ and τ because it helps us to distinguish between the set of elements and properties.

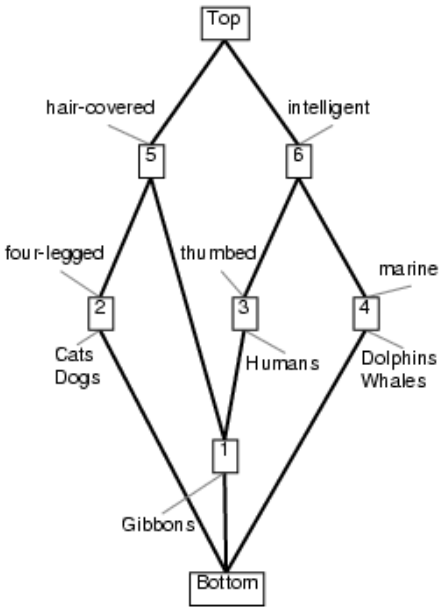


Figure 3.1: The lattice of the mammals example with classical notation.

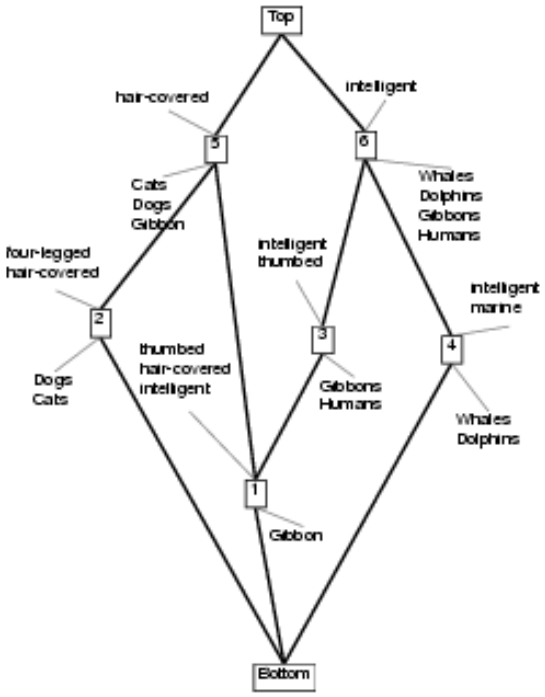


Figure 3.2: The lattice of the mammals example with complete notation.

The results of $c_3 \sqcap c_5$ and $c_1 \sqcup c_2$ from the mammal example are:

$$\begin{aligned} c_3 \sqcap c_5 &= (\{Gibbons, Humans\}, \{intelligent, thumbed\}) \\ &\quad \sqcap(\{Cats, Gibbons, Dogs\}, \{hair\text{-}covered\}) \\ &= (\{Gibbons\}, \sigma(\{Gibbons\})) \\ &= (\{Gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\ &= c_1 \end{aligned}$$

$$\begin{aligned} c_1 \sqcup c_2 &= (\{Gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\ &\quad \sqcup(\{Cats, Dogs\}, \{hair\text{-}covered, four\text{-}legged\}) \\ &= (\tau(\{hair\text{-}covered\}), \{hair\text{-}covered\}) \\ &= (\{Cats, Dogs, Gibbons\}, \{hair\text{-}covered\}) \\ &= c_5 \end{aligned}$$

Generalizing, the fundamental theorem for concept lattices [WILL 81] relates sub-concepts and super-concepts as follows:

$$\bigsqcup_{i \in I}(X_i, Y_i) = \left(\tau\left(\bigcap_{i \in I} Y_i\right), \bigcap_{i \in I} Y_i\right).$$

The significance of the theorem is that the least common superconcept (or *join*) of a set of concepts can be computed by intersecting their intents, and by finding the common elements of the resulting intersection. Equivalently we have defined the *meet*:

$$\bigsqcap_{i \in I}(X_i, Y_i) = \left(\bigcap_{i \in I} X_i, \sigma\left(\bigcap_{i \in I} X_i\right)\right).$$

In [SIFF 97], the superconcepts are defined as *covers* and the subconcepts are defined as *subordinates*. A concept d *covers* concept c if $c \sqsubseteq d$ and there is no concept e such that $c \sqsubseteq e \sqsubseteq d$. If d covers c, we say "c is covered by d". The set of covers of concept c, denoted by *covs(c)*, is the set of concepts d such that d covers c. The set of concepts *subordinates* to d, denoted by *subs(c)*, is the set of concepts c such that $c \sqsubseteq d$.

From the computation of the concepts, two *special* concepts are also introduced in the concept lattice. Given a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$, the two following concepts are calculated:

$$\begin{aligned} top &= (\tau(\emptyset), \sigma(\tau(\emptyset))) = (\mathcal{E}, \emptyset) \\ bottom &= (\tau(\sigma(\emptyset)), \sigma(\emptyset)) = (\emptyset, \mathcal{P}) \end{aligned}$$

The *top concept* reflects that none of the properties fit to *all* elements, and the *bottom concept* reflects that none of the elements fit to *all* properties. This means, that in the case of *top concept*, there is no column with crosses for all the elements in the table $\mathcal{T}$; and in the case of *bottom concept*, there is no row with crosses for all the properties in the table $\mathcal{T}$.

Concepts Labels in the Concept Lattice

There are two alternatives for labeling the concepts. Given a concept $c = (E, P)$, the label $l(c)$ can be defined as:

- $l(c) = (extent(c), intent(c)) = (E, P)$. This means, that the label lists all the elements and properties calculated for the concept.
- $l(c) = (E_n, P_m)$, if c is the *largest* concept with $p(\in P_m)$ in its intent, and c is the *smallest* concept with $e(\in E_n)$ in its extent. The (unique) lattice node labeled with p is denoted $\gamma(p) = \bigvee\{c \in \mathcal{L}(\mathcal{C}) \mid p \in int(c)\}$, and the (unique) lattice node labeled with e is denoted $\mu(e) = \bigwedge\{c \in \mathcal{L}(\mathcal{K}) \mid e \in ext(c)\}$. μ is often called contingent of a concept.

Both notations are useful, and the choice which of the two should be used depends on the case study to analyze. In the first case, where you have *extent* and *intent* of each concept, every node has all the related information. Differently in the second case, where you have $\gamma(c)$ and $\mu(c)$, the exclusive information about the concept is highlighted but if we want to see the complete information, we have to *navigate* through the cover and subordinate concepts. Experienced FCA users prefer the second notation because it is more clear.

3.2.3 Algorithms to Build the Concepts

There are several algorithms for computing the concepts for a given context, such as [SIFF 97], [GODI 98] or [SNEL 98a]. We describe two main algorithms: a simple bottom-up one introduced in [SIFF 97] – just to understand easily how the concepts are built – and the most efficient one introduced in [GANT 99]. Although there are a set of algorithms available depending on the needs of the case studies to analyze, we introduced these two ones because they are ones implemented in *ConAn* framework.

Bottom-Up Algorithm

Given a set of elements E, the smallest concept with extent containing E is $(\tau(\sigma(E)), \sigma(E))$. Thus, the bottom of the concept lattice is $(\tau(\sigma(\emptyset)), \sigma(\emptyset))$ – the concept consisting of all elements (often the empty set) that have all the attributes in the context relation.

The initial step of the algorithm is to compute the bottom of the concept lattice. The next step is to compute *atomic* concepts – smallest concepts with extent containing each of the elements treated as a singleton set. The computation of the atomic concepts in the mammal

example is the following list:

$$\begin{aligned}
\tau(\sigma(\{Cats\}) &= \tau(\{four\text{-}legged, hair\text{-}covered\}) = \{Cats, Dogs\} \\
\tau(\sigma(\{Dogs\})) &= \tau(\{four\text{-}legged, hair\text{-}covered\}) = \{Cats, Dogs\} \\
\tau(\sigma(\{Dolphins\}) &= \tau(\{intelligent, marine\}) = \{Dolphins, Whales\} \\
\tau(\sigma(\{Gibbons\}) &= \tau(\{hair\text{-}covered, intelligent, thumbed\}) = \{Gibbons\} \\
\tau(\sigma(\{Humans\}) &= \tau(\{intelligent, thumbed\}) = \{Humans, Gibbons\} \\
\tau(\sigma(\{Whales\}) &= \tau(\{intelligent, marine\}) = \{Dolphins, Whales\}
\end{aligned}$$

It is obvious that several calculations result in the same set of elements, meaning that these elements form the *extent* of the same concept, for example the elements *Cats* and *Dogs* in the first two calculations.

The algorithm then closes the set of atomic concepts under join: Initially, a work-list is formed containing all pairs of atomic concepts (*c'*, *c*) such that $c \not\sqsubseteq c'$ and $c' \not\sqsubseteq c$. While the work-list is not empty, remove an element of the work-list (c_0, c_1) and compute $c'' = c_0 \sqcup c_1$. If c'' is a concept that is yet to be discovered then add all pairs of concepts (c'', c) such that $c \not\sqsubseteq c''$ and $c'' \not\sqsubseteq c$ to the work-list. The process is repeated until the work-list is repeated. The iterative process of the concept-building algorithm can be found in the Table 3.3.

Ganter Algorithm

The *bottom-up algorithm* becomes awkward for larger contexts, since it requires consulting the list again and again. We describe the *Ganter algorithm* [GANT 99] which is faster for generating all the extents. This algorithm only uses the closure operator $A \rightarrow A''$ of the context, i.e. it is an *algorithm for the generation of all closures of a given closure operator*. Following with the notation used in this report, the closure operator of $A \subseteq \mathcal{E}$ = $\tau(\sigma(A))$.

First of all we consider the set of all subsets of $\mathcal{E}$ to be *in lexicographical order*. In our specific example, the set of animals ordered lexicographically is $\{Cats, Dogs, Dolphins, Gibbons, Humans, Whales\}$. For sake of simplicity we assume that $\mathcal{E}$ = $\{1, 2, ..., n\}$. A subset $A \subseteq \mathcal{E}$ is called *lexicographically smaller* than a subset $B \neq A$ if the smallest element which distinguishes A and B belongs to B. Formally:

$$A < B :\Leftrightarrow \exists_{i \in B \setminus A} \quad A \cap \{1, 2, ...i-1\} = B \cap \{1, 2, ...i-1\}$$

This defines a linear strict order on the powerset ($\mathcal{P}(\mathcal{E})$), i.e., for subsets $A \neq B$ always holds $A < B$ or $B < A$. The aim of the following is to find for an arbitrary given set $A \subseteq \mathcal{E}$ the extent that is smallest after A with respect to this lexicographical order. If we have solved this, we can obviously generate all extents as follows: The lexicographical smallest concept extent is $\tau(\sigma(\emptyset))$. The other extents are found incrementally by determining the one which is lexicographically closest to the last extent found. In the end, we obtain the lexicographical largest extent, namely $\mathcal{E}$.

$$
\begin{aligned}
c_0 &= (\{Cats, Dogs\}, \{hair\text{-}covered, four\text{-}legged\}) \\
c_1 &= (\{Gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\
c_2 &= (\{Dolphins, Whales\}, \{intelligent, marine\}) \\
c_3 &= (\{Gibbons, Humans\}, \{intelligent, thumbed\}) \\
Worklist &= [(c_0, c_1), (c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3)] \\
c_4 &= c_0 \sqcup c_1 = (\{Cats, Gibbons, Dogs\}, \{hair\text{-}covered\}) \\
Worklist &= [(c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_2 &= \top = (\{Cats, Gibbons, Dogs, Dolphins, Humans, Whales\}, \emptyset) \\
Worklist &= [(c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_3 &= \top \\
Worklist &= [(c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_5 &= c_1 \sqcup c_2 = (\{Gibbons, Dolphins, Humans, Whales\}, \{intelligent\}) \\
Worklist &= [(c_2, c_3), (c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_3 &= c_5 \\
Worklist &= [(c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_4 &= \top \\
Worklist &= [(c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_3 \sqcup c_4 &= \top \\
Worklist &= [(c_0, c_5), (c_4, c_5)] \\
c_0 \sqcup c_5 &= \top \\
Worklist &= [(c_4, c_5)] \\
c_4 \sqcup c_5 &= \top \\
Worklist &= \emptyset
\end{aligned}
$$

Table 3.3: Work-list of the Bottom-up algorithm

To make this precise, we define for $A, B \subseteq \mathcal{E}$, $i \in \mathcal{E}$,

$$
\begin{aligned}
A <_i B : &\quad \Leftrightarrow i \in B \backslash A \; and \; A \cap \{1, 2, ...i-1\}. \\
A \oplus i := &\quad \tau(\sigma((A \cap \{1, 2, ...i-1\}) \cup \{i\}))
\end{aligned}
$$

The new operator $\oplus$ leads to a new theorem that shows how we can find the concept extent we are looking for.

Theorem 1 *The smallest concept extent larger than a given set $A \subset \mathcal{E}$ (with respect to the lexicographical order) is*

$$A \oplus i,$$

i being the largest element of $\mathcal{E}$ with $A <_i A \oplus i$.

Algorithm for generating all extents of a given context $(\mathcal{E}, \mathcal{P}, \mathcal{I})$: The lexicographical smallest extent is $\tau(\sigma(\emptyset))$. For a given set $A \subset \mathcal{E}$ we find the lexicographically next extent by checking all elements $i \in \mathcal{E} \backslash \mathcal{A}$, starting from the largest one and continuing in a descending order until for the first time $A <_i A \oplus i$. $A \oplus i$ then is the "next" extent we have been looking for. Following with our mammal example, Table 3.4 shows the application of the algorithm to calculate the extents. For the sake of simplicity, we consider that each number 1..6 represents a mammal, following the same correspondence: {1: Cats, 2: Dogs, 3: Dolphins, 4: Gibbons, 5: Humans, 6: Whales}.

Step	i	New Extent	Set of Extents
1			$\emptyset$
2	4	{4}	$\emptyset$, {4}
3	5	{4,5}	$\emptyset$, {4},{4,5}
4	3	{3,6}	$\emptyset$, {4},{4,5},{3,6}
5	4	{3,4,5,6}	$\emptyset$, {4},{4,5}, {3,6},{3,4,5,6}
6	1	{1,2}	$\emptyset$,{4},{4,5}, {3,6},{3,4,5,6},{1,2}
7	4	{1,2,4}	$\emptyset$,{4},{4,5}, {3,6}, {3,4,5,6},{1,2},{1,2,4}
8	3	{1,2,3,4,5,6}	$\emptyset$,{4},{4,5},{3,6},{3,4,5,6},{1,2},{1,2,4},{1,2,3,4,5,6}

Table 3.4: Calculation of the extents of the mammal example using Ganter algorithm

Because of the duality between elements and properties, the algorithm can be transferred without change to the intents; we only have to replace the set $\mathcal{E}$ by $\mathcal{P}$.

Algorithm to build the Lattice

So far, we have described two possible algorithms to build the concepts. But the concepts have a *complete partial order* and they form a *lattice*. The Algorithm 1 is the simplest algorithm that ConAn uses to build the lattice.

The algorithm is not the most efficient one because we have to *visit* each concept several times in the list of concepts. But it is the simplest one to build the lattice.

The next section shows how this mathematical theory can – and is – applied in software engineering.

1: $\mathcal{C} \leftarrow (\mathcal{E}, \mathcal{P}, \mathcal{I})$
2: $edges \leftarrow \emptyset$
3: $S \leftarrow concepts(C)$
4: **for all** $c_1 \in S$ **do**
5: **for all** $c_2 \in S - \{c_1\}$ **do**
6: **if** $c_1 < c_2$ and ($\nexists c_3 \in S - \{c_1, c_2\} : c_1 < c_3$ and $c_3 < c_2$) **then**
7: $edges \leftarrow edges \cup \{c_1 \rightarrow c_2\}$
8: **end if**
9: **end for**
10: **end for**

Algorithm 1: Algorithm to build the lattice.

3.3 FCA Applied in Software Engineering

FCA has typically been applied in the field of software engineering to support software maintenance and object-oriented class identification tasks. In [TILL 03a] the authors present a broad overview by describing and classifying academic papers that report the application of FCA to software engineering. This section summarizes this paper and presents typically formal context definitions for software engineering.

Tilley et al. classified 42 publications using FCA. The found categories are split in the categories: *Early phase activities* and *software maintenance*.

Early phase activities are all the activities that occur before the commencement of coding [TILL 03a].

- *Requirement analysis*
  FCA approaches in this category supports the user by gathering and organize (semi-)automatically the requirements (*e.g.* written in natural language become a draft of a possible class hierarchy) [ANDE 97][DÜ 98][DÜ 99][DÜ 00][RICH 02a] [RICH 02b][RICH 02c][RICH 02d][RICH 02e][BÖ 01].

- *Component retrieval software*
  Lindig describes a retrieval system that could be used for retrieving software components from a library indexed by keywords [LIND 95].

- *Formal specification*
  Fischer builds on the component retrieval work of Lindig, however, instead of using keywords, a formal specification that captures the behavior of a software component is used [FISC 98].

- *Visualizing Z specification via FCA*
  Z is a state based formal method that exploits the theory and first order predicate logic [SPIV 89]. In [TILL 03b] they use ToscanaJ to conceptually navigate and explore a Z specification using FCA and retrieve relevant parts.

All the publications around **Software maintenance** have a common thread – extracting understandable structures that organize the artifacts of software systems. The found categories:

- *Dynamic analysis*
  [BALL 99] examines test coverage while [BOJI 00] and [EISE 01a] [EISE 01b] recover software architecture related to use cases.

- *Application to legacy systems*
  Snelting et al. used FCA to analyze the preprocessor commands in legacy C programs in order to examine the configuration structure [SNEL 96][KRON 94][FUNK 95]. [VAN 99] and [KUIP 00] compare the use of FCA for grouping fields within a large legacy COBOL program to that of hierarchical clustering. [CANF 99] follows a similar approach but are interested in organizing a legacy COBOL system into components suitable for distribution via CORBA. The task deriving object-oriented models from legacy systems written in C has been considered by [SAHR 97], [SIFF 97] and [TONE 01]. The general approach is to consider C functions as formal elements and the properties as either commonly accessed data structures or fields within commonly used structures.

- *Reengineering class hierarchies*
  Snelting explains a mechanism to re-organize class hierarchies using FCA [SNEL 97] [SNEL 98a][SNEL 98b]. Schupp et al. detect design flaws (*e.g.* orthogonality or lacking refinement) with their approach [SCHU 02]. [GODI 93] consider a context where the formal elements are methods and the formal properties are classes. They can detect as well design flaws like missing inheritance links. [HUCH 99] consider a concept lattice generated with classes as formal elements and properties derived from method signatures, thus including information about parameter types and return values. Each concept is considered as a candidate for a Java interface. [ARÉV 03a] shows a way to identify undocumented behavioral dependencies in order to do a safe reengineering of the class hierarchies.

- *Conceptual analysis of software structure*
  [TONE 99] attempts to recover design patterns in source code using a context in which the formal elements are tuples of classes and the properties are relations among those classes. [ARÉV 03b] can detect different groups of classes which belong together. This groups help to get the main features of a class and understand them better.

This chapter provided a snapshot of where FCA has been applied to support software engineering activities. The majority of the reported work has been in the areas of detailed design and software maintenance where FCA has been applied to object-oriented reengineering and class identification tasks. In the next chapter we apply FCA on the detection of Software Patterns.

Chapter 4

Detecting Software Patterns with FCA

4.1 Introduction

This chapter shows how to detect Software Patterns using FCA. The first section gives an overview of this approach. In the second section we explain how to define the elements and properties we need to use FCA. The explanation of the post processes on the first results follows. The last section introduces how the patterns are set in relation to each other.

4.2 Overview of the Approach

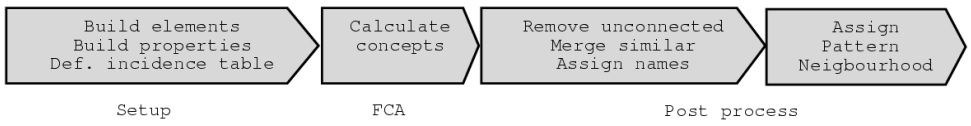


Figure 4.1: Overview of the approach

Figure 4.1 gives an overview of the approach. In the setup phase the elements and properties are built. These two sets form the table where the “crosses” are set. Setting a cross means that the tuple of the element and the property is added to the binary relation $\mathcal{I}$. In the second phase, the concepts are calculated. Those concepts become the patterns after the post process. The post process phase eliminates primary unwanted side-effects caused by the way elements and properties are built. Besides this, the found patterns are compared with a reference library to detect well-known patterns and name them. Furthermore the patterns are set in relation to each other.

4.3 Setup the Formal Context

This section explains the specific context $\mathcal{C}$ in order to detect Software Patterns. The **elements** $\mathcal{E}_o$ are tuples of classes which are analyzed. The length of these tuples is the *order* o. The **properties** $\mathcal{P}_o$ are *relations* inside one class tuple. Whenever such a relation $p_i \in \mathcal{P}_o$ within the tuple $e_j \in \mathcal{E}_o$ is fulfilled we add the relation (p_i, e_j) to the incidence table $\mathcal{I}$.

Figure 4.2 introduces a small pseudo application to better understand the theoretical explanations.This example consists of seven classes.

The crucial information are the relationships among the classes. We have two kind of class relationships:

- **Binary** relations R_B. These relations can be represented by a labeled pair $(C_i, C_j)_{Label}$, where C is the set of all the classes. Considering the class diagram in Figure 4.2, *e.g.* $(B, A)_{Sub} \in R_B$ is the relation *isSubclass* between B and A, and $(P, A)_{Acc}$ is the relation *accesses* between P and A.
- **Unary** relations R_U. These relations can be represented by a labeled class $(C_i)_{Label}$, *e.g. isAbstract*: $(A)_{Abstr}, (X)_{Abstr} \in R_U$.

Incidence Table 4.1 shows all the existing relations of order $o = 3$.

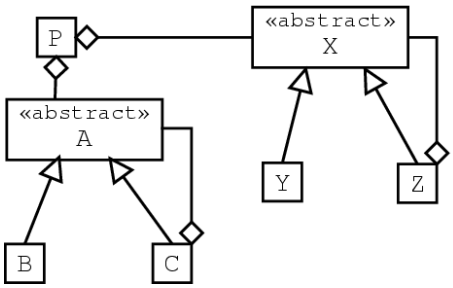


Figure 4.2: Example class diagram

Now that the formal context is defined, we have a closer look how elements and properties are *built*. The context building has to be called for each different order o, because a context in this approach is only valid for an order o, that contains the class tuples of length o.

4.3.1 Elements: Permutation of Classes

The elements are tuples of classes. The elements are all the permutations of the classes with the length of order o:

$$\mathcal{E}_o = \{(x_1, ..., x_o) \mid x_i \in C, 1 \leq i \leq o\}$$

	$(1,2)_{Sub}$	$(3,1)_{Sub}$	$(3,2)_{Sub}$	$(2,1)_{Acc}$	$(1,2)_{Acc}$	$(3,2)_{Acc}$	$(2,3)_{Acc}$
{C A P}	×				×	×	
{C A B}	×		×		×		
{Z X Y}	×		×		×		
{Z X P}	×				×	×	
{A P B}		×		×			
{A P X}				×			×
{Y X P}	×					×	

	$(1)_{Abst}$	$(2)_{Abstr}$	$(3)_{Abstr}$
{C A P}		×	
{C A B}		×	
{Z X Y}		×	
{Z X P}		×	
{A P B}	×		
{A P X}	×		×
{Y X P}		×	

Table 4.1: Order 3 context for the example in Figure 4.2

Tonella and Antoniol propose an inductive context construction algorithm to avoid the combinatorial explosion associated to generating all possible tuples of classes [TONE 99]. The underlying hypothesis is that design patterns consist of classes which are *all connected* together by their relations. In the initial step of the algorithm, all pairs of related classes are collected. In the inductive step, the class sequence from the previous iteration is augmented with all the classes having some relation with the classes in the sequence. Using the inductive context construction algorithm ensures that sequences representing disconnected graphs from the class diagram are disregarded.

In case of the example with the seven classes (Figure 4.2) all possible combinations of the class tuples of order $o = 3$ would lead to 210 elements[1], while the inductively constructed context contains only seven elements. The reduction in the number of elements is due to the presence of disconnected subgraphs in the class diagram, and the partial connectivity inside each connected graph. Such a reduction does not limit the ability of formal concept analysis to infer all possible Software Patterns [TONE 99]. We use exactly the same approach, but with a slightly difference: Compared with approach of Tonella, we just take *one* representative of each permutation whereas [TONE 99] also takes some combinations of tuples into account. This means that not only the tuple {C A P} is an element, but also {A P C} or {C P A}. Avoiding the combinations of a tuple makes the algorithm faster because we have even less elements as input. Having only one representative for all permutations causes unwanted side-effects *e.g.* the same patterns are found in two different concepts instead of one. These effects have to be eliminated in a later stage. To show the advantage of this

[1] $\binom{7}{3} 3! = 210$

difference we have a look on the quantitative effect in our example. Table 4.2 shows that the number of elements can be reduced what has an essential effect on the calculation time[2] of the concepts.

	our approach		[TONE 99]	
order	$\|\mathcal{E}\|$	time [s]	$\|\mathcal{E}\|$	time [s]
2	6	0.1	8	0.1
3	7	0.1	18	0.2
4	6	0.1	34	0.4
5	6	0.1	70	2.4
6	4	0.1	140	17.6
7	1	0.1	140	27.5
total	30	0.6	410	48.2

Table 4.2: Comparison between our inductive approach and the inductive approach from Tonella

4.3.2 Properties: Class Relations and Characteristics

The properties are all the possible combinations of a relation $C \times C$ inside a tuple $e_t \in \mathcal{E}$ together with the unary relations of each single class C:

$$\mathcal{P}_o = \{(i,j)_t \mid (x_i, x_j)_t \in R_B, 1 \leq i,j \leq o\} \cup \{(i)_t \mid (x_i)_t \in R_U, 1 \leq i \leq o\}$$

Each property has one or two indexes that refer to the position of the class to be analyzed inside the tuple. For example, the property $(3,2)_{Sub}$ applied on the element $\{$C A B$\}$ means that the class B is a subclass of the class A. Using indexes instead of names allows disjunct tuples to share common properties. In the example result (Table 4.3) the tuples $\{$C A B$\}$ and $\{$Z X Y$\}$ have the common properties $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$ and $(3,2)_{Sub}$.

4.3.3 Building the Incidence Table

To find out which properties hold, the algorithm has to consult the either directly the source code or a representative meta-model. In our approach we iterate simply over all the cells of the incidence table and mark the properties which hold.

4.4 FCA: Calculation of the Concepts

So far we specified all the prerequisites to start the calculation process of the concepts and lattice. There are several algorithms to calculate the concepts and its lattice [SIFF 97]. We

[2] Pentium 4, 1 GHz, 512 MB Ram

propose to use the Ganter algorithm [GANT 99], because it is one of the most efficient in terms of time performance. [KUZN 01]

The example of Figure 4.2 has ten concepts as result for the order $o = 3$. They are listed in Table 4.3.

In Figure 2.2, we have seen how a Composite Pattern looks like. Reducing and generalizing the structural information to *isSubclass*, *isAbstract* and *accesses* leads then to Figure 4.3. This simplified Composite Pattern is detected twice: {C A B} and {Z X Y} as concept 2.

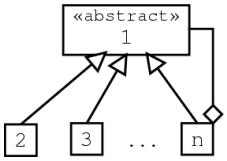


Figure 4.3: Structural aspects of the Composite Pattern

top	(*all elements* G, $\emptyset$)
8	({ {C A P}, {Z X P}, {C A B}, {Z X Y}, {Y X P} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$ })
7	({ {A P X}, {A P B} }, {$(2,1)_{Acc}$, $(1)_{Abstr}$ })
6	({ {C A P}, {Z X P}, {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$ })
5	({ {C A P}, {Z X P}, {Y X P} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(3,2)_{Acc}$ })
4	({ {A P B} }, { $(3,1)_{Sub}$, $(1)_{Abstr}$, $(2,1)_{Acc}$ })
3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abstr}$, $(3)_{Abstr}$, $(2,3)_{Acc}$})
2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })
bottom	($\emptyset$, *all properties* M)

Table 4.3: Concepts of the example in Figure 4.2

4.5 Post Filter on the Concepts

Once the concepts are calculated, each concept is a *candidate* for a pattern. But not all concepts are relevant. Therefore a post process on the concepts collection is needed to filter useless or uninteresting concepts. This section explains the filters.

4.5.1 Graph Representation of the Concept Intent

To make proper definitions we define a lemma which explains what the graph representation of the intent of a concept is. This lemma is domain specific to this work and can not be applied to general FCA.

Lemma 1 (Intent relation graph) *An intent relation graph is a graph which has as nodes all the indexes of the properties from the binary relations R_B. The edges of this graph are the binary relations R_B between the indexes.*

$$Nodes = \{n_i \mid 1 \leq i \leq o\}$$

$$Edges = \{(n_i, n_j) \mid (i, j) \in R_B\}$$

Considering Table 4.3, the intent graphs of the three concepts are shown in Figure 4.4. Considering the first diagram $c2$, the edge between node 1 and node 2 is from the property $(1, 2)_{Sub}$ or $(1, 2)_{Acc}$ and the second edge between node 2 and 3 is from the property $(3, 2)_{Sub}$. The edges are unlabeled and have no weights. As soon as at least one relation between two nodes holds, the edge exists. Note that the edges from the intent relation graph have nothing in common with the lattice edges.

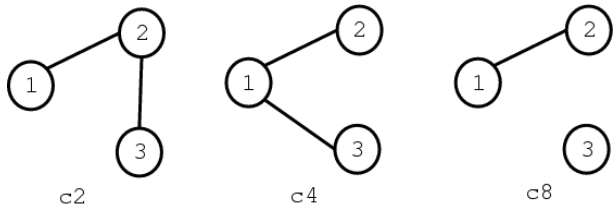


Figure 4.4: The intent graph of the second, fourth and eighth concept from Table 4.3

4.5.2 Removing Unconnected Patterns

A concept is only interesting when the intent (all properties which are for this concept true) is a set of structurally connected nodes.

Definition 1 (Connected Pattern) *A connected pattern is a pattern whose intent relation graph (Lemma 1) is fully connected.*

Unconnected patterns are determined by a lower order context. A context of the order o is computed, when order o patterns are looked for, while $o-1$, $o-2$, ... order contexts suffice for lower order patterns.

In the example result (Table 4.3) the following concepts are unconnected: 6, 7, 8 and the top concept.

4.5.3 Merging Equivalent Patterns

Suppose we have a system with the classes as shown in Figure 4.5. It might then happen that the ConAn black box finds the two concepts shown in Table 4.4.

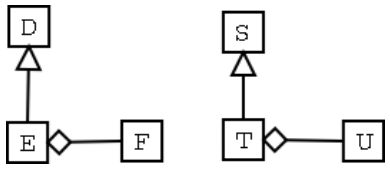


Figure 4.5: Twice the Adapter Pattern

c_1	({ {D E F} }, { $(2,1)_{Sub}$, $(3,2)_{Acc}$})
c_2	({ {T U S} }, {$(1,3)_{Sub}$, $(2,1)_{Acc}$ })

Table 4.4: Concepts of the example in Figure 4.5

Even though {D E F} and {T U S} are exactly the same pattern, the algorithm treats them separately. This happens because the elements are only the permutations without their combination. This means we just look at {D E F} and not to {D F E}, {E D F}, {E F D}, {F D E} nor {F E D}.

Definition 2 (Equivalent Patterns) *Two concepts, representing design patterns, are equivalent if a permutation of the indexes of the intent properties exists such that each property from the first concept can be transformed into a property of the second concept by that permutation, and vice versa* [TONE 99].

In our example we find a permutation $\alpha = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$, which transforms the tuple: {T U S} $\overset{\alpha}{\longmapsto}$ {S T U}. Concept c_2 can now be removed when the translated extent of this concept is added on concept c_1. This process is called **merging two concepts**.
In the main example (Table 4.3) the concept 4 and 5 are equivalent: The same permutation α from above translates the properties of concept 5 into those of concept 4. The pseudo code to merge equivalent patterns is presented in Algorithm 2.

Applying these two filters on the main example leads to four patterns presented in Table 4.5. The first three patterns are directly taken from the first three concepts. Pattern p_4 is merged from concept 4 and 5. The elements of concept 5 are translated into {P C A}, {P Z X} and {P Y X} and append to {A P B}.

p_4	({ {A P B}, {P C A}, {P Z X}, {P Y X}}, { $(3,1)_{Sub}$, $(1)_{Abstr}$, $(2,1)_{Acc}$ })
p_3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abstr}$, $(3)_{Abstr}$, $(2,3)_{Acc}$})
p_2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
p_1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })

Table 4.5: Resulting Patterns after the merging of equivalent patterns from the concepts of Table 4.3

1: $a := 1$
2: **while** $a \leq$ sizeof$(patterns)$ **do**
3: $b := b + 1$
4: **while** $b \leq$ sizeof$(patterns)$ **do**
5: find a permutation α which makes $pattern_a$ equivalent with $pattern_b$
6: **if** α exists **then**
7: Append the elements of $pattern_b$ translated by the invert permutation α to the $pattern_a$
8: remove $pattern_b$
9: $b := b - 1$
10: **end if**
11: $b := b + 1$
12: **end while**
13: $a := a + 1$
14: **end while**

Algorithm 2: Algorithm to merge equivalent patterns

4.5.4 Guessing Names for Patterns

Some of the detected patterns might be well known Design Patterns. To check this, we need a library of *named reference patterns*. In our implementation we provide this library by programming concrete instances of patterns. For the Composite Pattern an instance with the properties $(1,2)_{Sub}$, $(2)_{Abstr}$, $(3,2)_{Acc}$ and $(3,2)_{Sub}$ for the order $o = 3$ is stored as a reference (compare Figure 4.3). With the same algorithm from the previous section which detects candidates for the merging, the found patterns are assigned to the references and assigned name of the pattern. Thus the pattern p_2 from our example is referred as the Composite Pattern.

4.6 Pattern Neighborhood

One of the advantages of the FCA approach is that the found groups are *not isolated*. With the lattice they are set in relation to each other. A cover concept in the lattice has less properties, whereas a subordinate concept in the lattice has more properties. Thus we can define the idea of neighborhood. A higher neighbor c_2 of c_1 means that the intent of c_2 is a subset of the intent of c_1: $int(c_2) \subseteq int(c_1)$.

Definition 3 (Almost pattern) *An almost pattern* X *of a pattern* Y *is a pattern* X *which is a superconcept (direct cover neighbor in the lattice) of the pattern* Y.

The corresponding definition with the subordinate concepts:

Definition 4 (Overloaded pattern) *An overloaded pattern* X *of a pattern* Y *is a pattern* X *which is a subconcept (direct subordinate neighbor in the lattice) of a pattern* Y.

An overloaded pattern c_1 covers the properties of c_2.

Normally this information can directly be taken out from the lattice. The lattice of the example is shown in Figure 4.6.

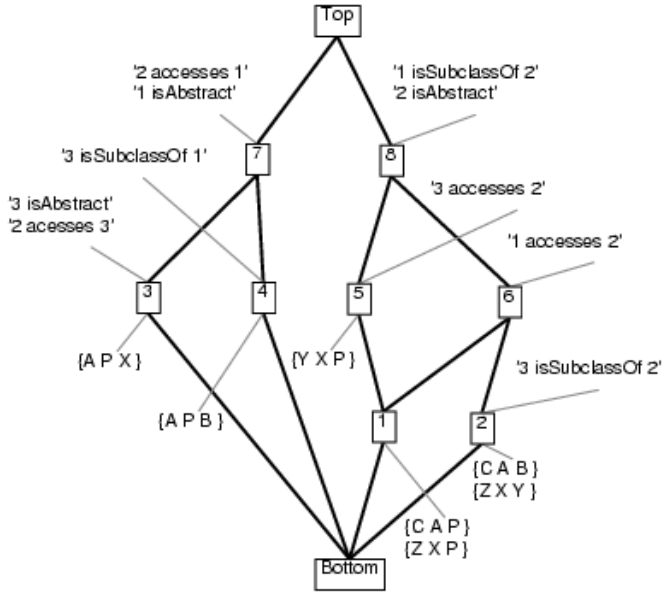


Figure 4.6: Resulting lattice of Incidence Table 4.1

Looking at this lattice, the concept 5 is an almost pattern of concept 1. The property $(1,2)_{Acc}$ is missed in the tuple $\{\text{Y X P}\}$ that it would fit the concept 5.

The problem with our concrete approach is that the side-effect of having *equivalent patterns* has to be taken into account here as well. As we have seen concept 4 is equivalent to concept 5. Therefore concept 4 is an almost pattern as well, or to be more precise the union of concept 4 and 5 (the pattern p_4 from Table 4.5) is the almost pattern. That is the reason why we do not calculate the classical lattice, but calculate a graph without the fundamental theorem for concepts (Section 3.2.2) instead. Whenever such an almost pattern connection is found we have to see if the elements of this overloaded pattern are already in the almost pattern. If not they have to be added. In our example, pattern p_4 is as well an almost pattern of p_2 but only when p_2 is translated with the permutation $\alpha = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$. Thus the elements of p_2 have to be added in p_4: $\{\text{C A B}\} \overset{\alpha}{\longmapsto} \{\text{A C B}\}$ and $\{\text{Z X Y}\} \overset{\alpha}{\longmapsto} \{\text{X Z Y}\}$.

Now the patterns reached their final state and are listed in Table 4.6.

p_4	({ {A P B}, {P C A}, {P Z X}, {P Y X}, {A C B}, {X Z Y}}, { $(3,1)_{Sub}$, $(1)_{Abstr}$, $(2,1)_{Acc}$ })
p_3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abstr}$, $(3)_{Abstr}$, $(2,3)_{Acc}$})
p_2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
p_1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })

Table 4.6: Final Patterns after applying the post filters on the concepts from Table 4.3

4.6.1 Linking Different Orders

Almost and overloaded patterns remain in the same order o. There are as well related patterns in the order $o-1$ and the order $o+1$. Patterns in a higher order have sub patterns from the lower order. They can be detected by subgraph matching techniques.

Definition 5 (Cover pattern) *C_2 is a cover pattern of the pattern c_1, if the intent relation graph (Lemma 1) of the pattern c_1 is a subgraph of the intent relation graph of c_2.*

Cover patterns are the connection links to the patterns in the order $o+1$. Note that a cover *pattern* has nothing in common with a cover *concept*. A cover concept would be in our terminology an almost pattern. This complication is caused from the fact that we do not look at one single lattice any more, but linking different lattices (one lattice for each order).

Definition 6 (Sub pattern) *C_2 is a sub pattern of the pattern c_1, if the intent relation graph (Lemma 1) of the pattern c_2 is a subgraph of the intent relation graph of c_1.*

Sub patterns are the connection links to the patterns in the order $o-1$. Linking the different orders of the patterns is made *after* applying the post process filter.

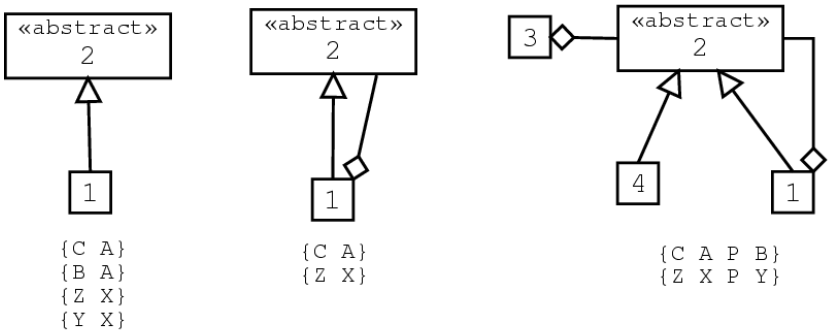


Figure 4.7: Sub and cover patterns of the Composite Pattern (p_2)

The first two diagrams in Figure 4.7 are the sub patterns of the Composite Pattern p_2 in the lower order $o=2$. The third one is the only cover pattern in order $o=4$.

The pattern neighborhood is now the union of all the above mentioned sets:

Definition 7 (Pattern Neighborhood) *A pattern neighborhood of a pattern c is the union of the almost, overloaded, cover and sub patterns of the pattern c.*

4.7 Complexity of the Approach

An important aspect of the approach is the complexity of the algorithms. The number of found concepts can be exponential in the size of the input context [KUZN 01]. The time complexity of the Ganter algorithm is $O(|\mathcal{E}|^2|\mathcal{P}|n)$, where n is the number of found concepts. Ganter has internal preparation phase (known as *polynomial delay*[JOHN 98]) which has an additional complexity of $O(|\mathcal{E}|^2|\mathcal{P}|)$ [KUZN 01]. The time complexity of the post process is $O(n^2)$, where n is again the number of concepts.

Chapter 5

Validation: Case Studies

5.1 Introduction

The validation of our approach and our tool *ConAn PaDi* is done based on three case studies. The first section explains how we instantiate those case studies. We have a look at three mid-sized Smalltalk applications: ADvance, SmallWiki and CodeCrawler. The results of these case studies are discussed in the last section.

5.2 Methodology for the Case Studies

This section explains the needed steps to analyze an application. The precondition is that all the needed components are loaded into the Smalltalk image: *ConAn PaDi*, ConAn framework, *Moose* and the application to analyze.

The following two steps have to be done:

1. Importing the application into the *Moose* meta-model. This step transforms the source code into its representative meta-model. The *ConAn DP Operator* has to be applied. This operator assigns all the additional information to the entities (*e.g.* classes, attributes, methods). This information is, for example, the possible type of an attribute. As Smalltalk is a typeless language, this information is not available from the source code directly and has to be assumed by special techniques. We used the method and tool described in [AEBI 03].

2. Starting the *ConAn PaDi* importer. This tool starts all the needed processes and the found patterns are then presented in the pattern browser (Figure 6.4).

Instead of importing a Smalltalk application into *Moose* it would as well be possible to load a meta-model from an application written in another language (*e.g.* Java, C++). It

can happen that this imported meta-model has not all the needed information for the properties (*e.g.* isSingleton), but this does not invalidate the whole approach, rather just those properties will never be true.

5.3 The Applications: ADvance, SmallWiki and CodeCrawler

The following three applications have been taken as case studies:

- **ADvance** is a system round-trip engineering tool from IC&C [1]. It is a multidimensional OOAD-tool for supporting object-oriented analysis and design, reverse engineering and documentation. ADvance is as well integrated in our tool *ConAn PaDi* where it shows the structure of a selected tuple.
- **SmallWiki** is a new and fully object-oriented wiki implementation in Smalltalk from Lukas Renggli[2]. It is highly customizable and easily extensible with new components, looks, servers, storage, etc.
- **CodeCrawler** is a language independent software visualization tool[3]. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [LANZ 99][DEME 99][DUCA 01b] [LANZ 01][LANZ 03].

After importing the classes into the meta-model *Moose*, we selected just those which belong to the namespace of the application. From all the classes we just looked at the instance side. Table 5.1 gives an overview of the quantity of the selected entities.

	ADvance	SmallWiki	CodeCrawler
Number of classes	167	100	81
Number of methods	2719	1072	1077
Lines of code	14466	4347	4868
Computation time	$\sim$2 days	$\sim$2 hours	$\sim$30 min

Table 5.1: Statistical overview of the cases

We want to analyze the impact of different sets of properties; *ConAn PaDi* supports this by letting the user define different filters. The used filters are shown in Table 5.2. The advantage of these filters is that once the patterns are in the repository it just takes some seconds to apply a filter instead of restarting the whole calculation process which took up to two days on a home PC [4]. The calculation time is as well the reason why we took only the patterns up to the order $o = 4$. In the case of ADvance the patterns up to order $o = 3$ were calculated in less than one hour where as the order $o = 4$ needed two days! The calculation time depends on how many concepts are found (Section 4.7).

[1] http://www.io.com/$\sim$icc/
[2] http://c2.com/cgi/wiki?SmallWiki
[3] http://www.iam.unibe.ch/$\sim$scg
[4] Pentium 4, 1 GHz, 512 MB Ram

Filter A	Filter B
isSubclass hasAsAttribute isAbstract	isSubclass hasAsAttribute

Table 5.2: Used filters

The quantitative impact of the filters is seen in Table 5.3. A filter with less properties gives a clearer image of the situation. Applying the Filter B with less properties heavily reduces the number of different patterns whereas the found patterns in total are much less reduced. This leads to patterns which have more tuples as elements. The patterns of Filter A are too "noisy". CodeCrawler, for example, has in the order $o = 3$ in the average 13 tuples per pattern for the Filter A (431 / 32) whereas the Filter B has an average of 21 tuples per pattern (300 / 14).

		ADvance		SmallWiki		CodeCrawler	
o		Filter A	Filter B	Filter A	Filter B	Filter A	Filter B
2	# different patterns	12	5	8	4	7	3
	# patterns in total	215	181	138	114	116	85
3	# different patterns	57	32	37	17	32	14
	# patterns in total	1103	907	471	402	431	300
4	# different patterns	329	218	190	93	110	58
	# patterns in total	7521	6093	2293	2039	1423	983

Table 5.3: Filter statistics

With 34 core classes of CodeCrawler we analyzed patterns of higher order $o > 4$. The statistics are in Table 5.4. Unfortunately this calculation takes one week on a home PC.

order o	2	3	4	5	6	7	8
# different patterns	6	24	64	157	335	650	1157
# patterns in total	45	94	209	461	954	1850	3375

Table 5.4: Patterns of higher order of a set of core classes from CodeCrawler

To better compare the three cases we selected eight reference patterns which are introduced in Table 5.5. *Subclass Star* is a tuple where one class has all the others as subclass. In the *Subclass Chain* the classes form an inheritance chain, whereas in the *Attribute Chain* the classes form an access chain. *Attribute Star* is a pattern with a class which is used as attribute in all the other classes from the tuple. The next four patterns (*Facade*, *Composite*, *Adapter* and *Bridge*) have all names from the collection of [GAMM 95]. It is important to see that they are simplified to the used structural aspects of inheritance, aggregation and abstractness of a class. If our tool identifies a found pattern with such a reference pattern it just means that it *could* be a candidate for this pattern. The letter A in a box means that the class should be abstract.

Table 5.6 shows all the found patterns of those eight references for the order $o = 2, 3, 4$. Most of the patterns appear twice: Once in the first line (*e.g. Composite*) where the reference pattern lacks the property *isAbstract*; whereas in the second line the *isAbstract* property is taken into consideration. As Filter B has no property *isAbstract* it is obvious that no patterns containing an *isAbstract* property can be found.

One interesting observation is the two zeros marked with a star: *Subclass Chain* and *Attribute Chain* of order $o = 4$ of the CodeCrawler application. Applying Filter A no *Subclass Chain* nor *Attribute Chain* of order $o = 4$ is found. Nevertheless Filter B detects 12 instances of *Subclass Chain* and 15 instances of *Attribute Chain*. Filter A does not detect those patterns because CodeCrawler has no chain with *exactly* at the beginning an abstract class and the rest of the chain consists of non abstract classes. Just one representative with an abstract class on top would be enough that the FCA approach would detect the rest of the 12 (resp. 15) patterns. This effect shows that having too many properties can be counterproductive and the basic patterns can not be detected because there is too much "noise".

In the next section we start the discussion how useful these results are.

5.4 Discussion of the Results

We tried to use this approach to reach the objectives of reverse engineering mentioned in Section 2.5. We settled down concrete goals derived from the six objectives listed by [CHIK 90]. We want to verify if our goals could be reached:

1. *Gain an overview of the system structure*
   With the tool *ConAn PaDi* it is possible to have a list of all the classes from a system. To gain an overall overview the patterns of higher order turned out to be more interesting, but they need a long computation time. Traditionally class diagrams and use cases are more useful for the overview and are computed much faster.

2. *Locate and understand relations such as inheritance, accesses and invocations*
   The filtering possibilities of *ConAn PaDi* are useful to have a look at one single class or a set of classes. Our tool is adequate to find out the relations of those filtered classes. For example we have a look at the CodeCrawler application: Just looking at the patterns containing the class `CCNodePlugin` we see that the following classes are related with this class: `CCItemPlugin`, `CCCompositeNodePlugin`, `CCFAMIXNodePlugin` are inheritance relations. There are no attribute relations.

3. *Detect class dependencies*
   As all the different relations (inheritance, access, invocation) are shown, the dependencies derived from those relations are then available. Looking again at CodeCrawler, we see that *e.g.* the class `CCTool` cannot have a lot of dependencies because this class is in none of the patterns, whereas `CCNodePlugin` is in 47 patterns up to order $o = 4$ and must therefore have several dependencies.

4. *Identify the possible presence of classical design patterns*
   Candidates for classical design patterns are found. Not all of them turn out to be a real design pattern. The reasons for the misinterpretation are:

   - Not all the properties are absolute reliable. For example, the extraction of the type of an attribute is based on a heuristic [AEBI 03].
   - The false candidate has just by chance the same properties. Mainly with the Facade, Adapter and Bridge pattern the intention of the developer could be another one and not the one of the proposed pattern. As example we have a look at the Bridge pattern of the order $o = 3$ in Table 5.5: That a class has a subclass and accesses another class means that it is a candidate for a Bridge, but there can be a lot of other reasons why this tuple holds this structural aspects.

   To decide which are real classical design pattern detailed knowledge about the application is needed. It can not be decided with the information provided by *ConAn PaDi*. We were just able to detect structural Design Patterns (explained in Section 2.3) such as: *Adapter*, *Bridge*, *Composite* or *Facade*. We could just detect these patterns because we have primarily *structural* information about the entities (classes, methods, attributes, *etc*). Extracting only structural information is not enough to infer behavioral patterns [TONE 99].

5. *Identify the neighborhood of a pattern*
   The neighborhood can be analyzed by navigating through the almost, overloaded, sub and cover pattern. This can be important to detect all candidates for a classical pattern. For example, we consider the *Abstract Composite* pattern of order $o = 3$ of the ADvance application. Applying Filter A *ConAn PaDi* detects two *Abstract Composite* patterns, but in the neighborhood we find four more *Composite* patterns without an abstract composite root.

6. *Find candidate classes for restructuring*
   In our case studies we could not find a candidate for restructuring, because our analysis was made on structural information. Restructuring needs control flow information and detailed information about the method bodies [TICH 01]. Furthermore a lot of domain specific knowledge is needed. Nevertheless we believe that future investigation with our tool such a detection is possible, when the above mentioned information is available and taken into consideration.

7. *Identify new unknown and undescribed patterns*
   As our approach with FCA detects *any* kind of patterns we found plenty of "new" patterns, meaning that they are not referred in literature. Whether those patterns are useful and make sense as Design Patterns is another issue. If we see in Table 5.3 the case study ADvance for the order $o = 4$, we detect 329 different patterns. The frequency of several patterns is shown in Table 5.6. We see that the Facade is detected 627 times and the Bridge only 20 times. The accumulated elements of the pattern is shown again in Table 5.3. All the elements of all the patterns for order $o = 4$ of

the ADvance application is 7521. A reason for this high amount of patterns is the following. Consider that an application has a class C_0 with five subclasses $C_{1..5}$. In the order $o = 5$ a *Subclass Star* will be detected once. But for example in the order $o = 3$, our approach detects 10 subclass stars: $\{C_0\ C_1\ C_2\}$, $\{C_0\ C_1\ C_3\}$, $\{C_0\ C_1\ C_4\}$, ..., $\{C_0\ C_4\ C_5\}$.

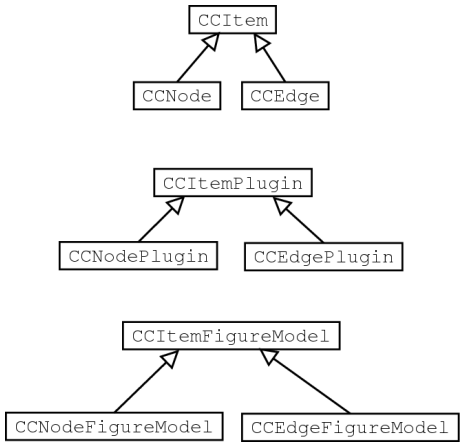


Figure 5.1: Three *Subclass Stars* of CodeCrawler

8. *Understand the class roles*
   With the gained information about the class relations it is possible to guess its role. In CodeCrawler we detect in the order $o = 3$ three *Subclass Stars* shown in Figure 5.1. The assumption that these are three parallel structures cooperating together are proved by future investigation.

9. *Identify coding styles*
   We have seen that frequency and the existence of a pattern in a system is besides domain specific issues, as well an issue of coding style. In our case studies we have seen that CodeCrawler has a lot of *Subclass Star* and *Facade*, whereas SmallWiki has a lot of *Attribute Chain* and *Attribute Stars*. ADvance is the only application with the *Composite* pattern.

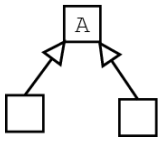
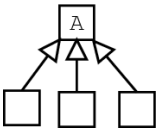
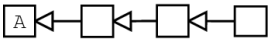
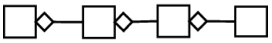
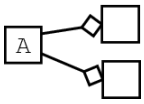
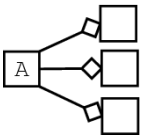
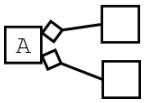
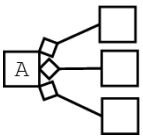
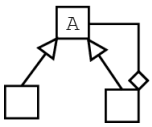
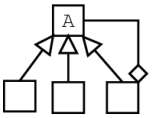
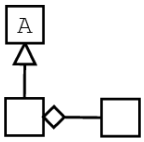
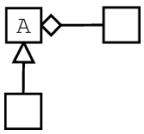
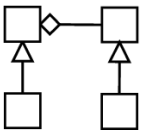
	$o = 3$	$o = 4$
Subclass Star		
Subclass Chain		
Attribute Chain		
Attribute Star		
Facade		
Composite		
Adapter		
Bridge		

Table 5.5: Structure of investigated patterns

		ADvance		SmallWiki		CodeCrawler	
o	Pattern	Filter A	Filter B	Filter A	Filter B	Filter A	Filter B
2	Subclasses	95	95	84	84	57	57
	Attributes	80	80	28	28	26	26
3	Subclass Star	271	271	118	118	140	140
	Abstract Subclass Star	46	-	12	-	22	-
	Subclass Chain	44	44	62	62	28	28
	Abstract Subclass Chain	10	-	10	-	11	-
	Attribute Chain	108	108	39	39	25	25
	Facade	214	214	23	23	42	42
	Abstract Facade	0	-	0	-	15	-
	Attribute Star	44	44	24	24	9	9
	Abstract Attribute Star	3	-	3	-	1	-
	Composite	6	6	0	0	0	0
	Abstract Composite	2	-	0	-	0	-
	Adapter	32	32	13	13	4	4
	Abstract Adapter	13	-	1	-	1	-
	Bridge	37	37	44	44	19	19
	Abstract Bridge	6	-	5	-	12	-
4	Subclass Star	1073	1073	135	135	313	313
	Abstract Subclass Star	87	-	5	-	15	-
	Subclass Chain	12	12	31	31	0*	12
	Abstract Subclass Chain	1	-	10	-	3	-
	Attribute Chain	137	137	46	46	0*	15
	Facade	627	627	13	13	56	56
	Abstract Facade	0	-	0	-	20	-
	Attribute Star	15	15	22	22	0	0
	Abstract Attribute Star	1	-	3	-	0	-
	Composite	3	3	0	0	0	0
	Abstract Composite	1	-	0	-	0	-
	Bridge	20	20	43	43	6	6

Table 5.6: Investigated Patterns

Chapter 6

ConAn PaDi, a Pattern Browser

6.1 Introduction

To validate the detection of Software Patterns with FCA, we have written a tool, called *ConAn PaDi*, a pattern browser. The name is an acronym for *Con*cept *An*alysis *Pa*ttern *Di*splayer. This tool is based on the reengineering environment *Moose* and uses the *ConAn* framework to calculate the concepts, which are introduced in the first two sections. The next section takes a close look at the *Fish Eye View* technique. The last two sections explain the user interface and the tool architecture.

6.2 The meta-model: Moose

The meta information is taken from the FAMIX meta-model. This section gives an introduction to *Moose*, a reengineering environment, implemented in Smalltalk.

In the past few years the Software Composition Group (SCG) at the University of Bern has been involved in a number of research projects in the field of software re- and reverse engineering. In the FAMOOS[1] project European partners came together to build a number of tool prototypes to support object-oriented reengineering.

To avoid equipping the tool prototypes with parsing technology for different programming languages, a common information exchange model with language specific extensions is specified. This model is named FAMIX (**FAM***OOS* **I***nformation E***X***change Model*) [DEME 01][TICH 01].

Moose is a reengineering research platform implemented in VisualWorks Smalltalk [DUCA 00][DUCA 01a][TICH 01]. It has been developed during the FAMOOS project to reverse engineer and re-engineer object-oriented systems. It consists of a repository to store models of source code. The models are stored based on the entities defined in

[1] `http://www.iam.unibe.ch/~famoos/`

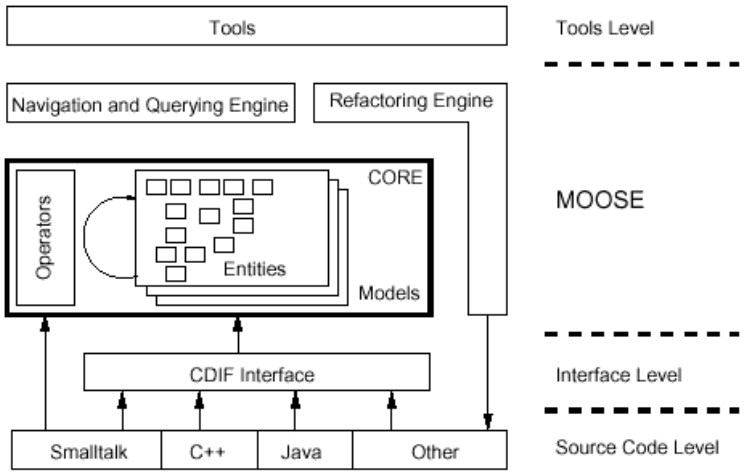


Figure 6.1: *Moose* architecture.

FAMIX. The software analysis functionality of *Moose* is language independent. The FAMIX models can be loaded from and stored to files. Apart from the repository, there are other features implemented to support reverse engineering activities:

- a parser for Smalltalk code
- an interface to load and store information exchange files
- a software metrics calculation engine
- an interface for additional tools to browse and visualize stored entities

6.3 ConAn: a Framework for FCA

ConAn is a general, extensible framework for FCA implemented in VisualWorks 7.0 by Gabriela Arévalo and Frank Buchli from the Software Composition Group of the University of Bern. Its name is an acronym for *Con*cept *An*alysis. It allows the user to define any kind of elements and their properties, and calculate the concepts and the lattice. The user can analyze the results provided by the framework. Four GUI tools in this framework support the user: *Formal Context Editor, Concepts Viewer, Concept Crawler* and *Fish Eye Viewer*. The framework can be used as a ready-to-use tool with an interactive GUI or as a white-box framework. The second alternative focuses on users who want to integrate FCA in their applications. Thus, automated or semi-automated analysis is possible.

Two main features distinguish this framework: *extensibility* and *encapsulation of the mathematical theory*.

Regarding the **extensibility**,

- The framework can be applied to any domain, and it is not limited only to software understanding.
- Any Smalltalk objects can be used as elements or properties. This makes the framework easy to adapt on a specific domain, where the input can be a result from a pre-process.
- New algorithms can be implemented and easily incorporated in the framework
- The user can develop new post-process filters to be able to better analyze the results
- The user is not obliged to restrict himself to the existing tools, new tools can be developed on top of the framework.

Regarding the **encapsulation of the mathematical theory**, the user is never concerned with the mathematical background of the lattice theory, which is the basis of this framework, and is implemented as a *black box* inside the framework. The user only has to know the public interfaces of two classes: the application facade and the class representing the concepts.

The rest of this section introduces the framework itself. First an overview of the different phases is given and secondly the provided GUIs are presented.

The process from the input to the output provided by this framework is defined by 5 phases as shown in Figure 6.2:

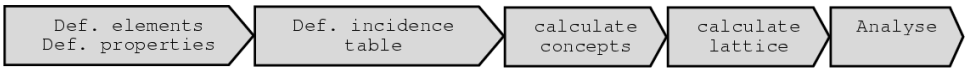


Figure 6.2: Overview of the different phases in the ConAn framework

- *Definition of the elements and properties*
  The user has to define which are the elements and the properties of the context. This task can be done using the *Formal Context Editor* or by programming it. In the first case, the user interacts with an empty table where the elements and the properties are defined as labels. In the second case, the user has to provide the Smalltalk objects that represent elements and properties.
- *Building the incidence relation table*
  The specific domain knowledge of the user is needed in this phase. The user has to define the incidence relation table using the *Formal Context Editor* or using the framework interface.

- *Calculation of Concepts*
  In this phase, the concepts are calculated with the algorithm that the user has previously chosen. So far, the framework provides two possible choices: *Bottom-up algorithm* [SIFF 97] and *Ganter algorithm* [GANT 99].

- *Calculation of Lattice*
  The lattice is built according to the complete partial order of the concepts. So far, the framework only provides one algorithm [GANT 99]. This phase is optional, meaning that for some case studies, the concepts without their lattice are sufficient to analyze the results. Obviously, all the support-tools based on the lattice information cannot be used in the next phase.

- *Analysis of the Results*
  The framework provides three tools which might help the user to analyze the results.

The following paragraphs introduce the three tools from the ConAn framework which support the user in analyzing the results:

- *Concepts Viewer*
  This tool allows the user to see the elements and the corresponding properties. It is also possible to see the intent and the extents of each concept, and its subordinates and cover concepts.

- *Concept Crawler*
  This tool allows the user to display the lattice as a graph. The user can see the intent and extent of each concept, or a reduced version where all the elements and properties are just appearing once (read more in Section 3.2.2). One additional feature of this tool is the possibility of applying some metrics to the nodes (graphical representation of the concepts). This tool is built on top of CodeCrawler [2] [LANZ 99].

- *Fish Eye View*
  If we showed to the user the *complete* lattice, he or she would be sometimes overwhelmed by the information. The idea is to have something like a hyperbolic browser [LAMP 95]. To cope with this problem the ConAn framework provides a special view called the *Fish Eye View*. The name fish eye comes from the special fish eye camera lenses which focuses on the center and marginalizes the rest [FURN 81]. This tool gives the focused view on only one concept. The lattice edges are used as navigation links to browse the whole lattice. The results provided by the framework are not necessarily the final ones, this means that in this phase we can also have a domain-specific post-processing shaping the results.

[2] http://www.iam.unibe.ch/∼scg

6.4 Fish Eye View on a Pattern

This section explains how the *Fish Eye View* is applied in *ConAn PaDi*. The view consists of four lists.

- *Extent:* All the elements of a concept.
- *Intent:* All the properties of a concept.
- *Generalization:* The upper edges in the lattice from the concept.
- *Specialization:* The lower edges in the lattice from the concept.

These lists for the fifth concept of the example (lattice in Figure 4.6 before the post process) are shown in the screen-shot in Figure 6.3.

The two lists in the middle are the extent (elements) and intent (properties) of the concept 5. The item in the top list is the edge to concept 1. This is a specialization, because it adds the property $(1,2)_{Acc}$. The highlighted elements {C A P} and {Z X P} are the elements which remain by this specialization. The item in the bottom list is the edge to concept number eight. It takes away the property $(3,2)_{Acc}$. The remaining properties $(2)_{Abstr}$ and $(1,2)_{Sub}$ are highlighted.

Clicking on an item of the generalization or specialization list, jumps to this concept. Like this the user can navigate through the concept lattice.

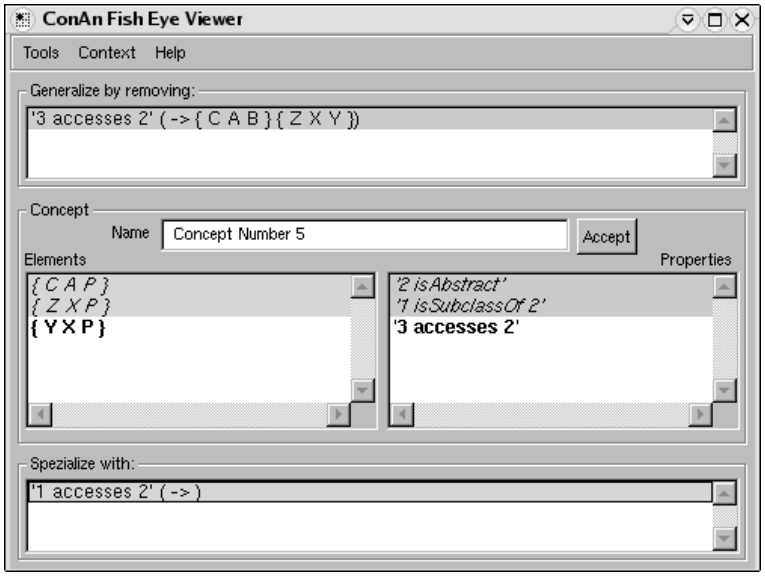


Figure 6.3: Implementation of the *Fish Eye View* in ConAn

Having more navigation dimensions (sub and cover patterns) and the merging of equivalent pattern requires an enhancement of this view. This done by adding two more lists for the

sub and cover pattern shown in Figure 6.4.

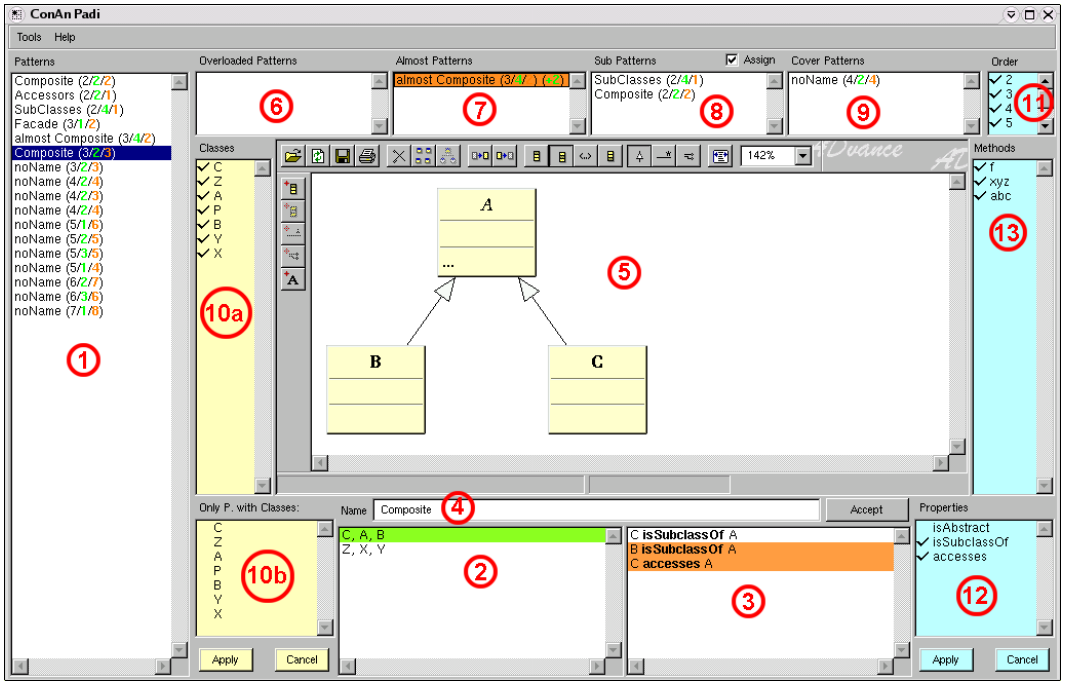


Figure 6.4: *ConAn PaDi* with the result from the classes of Figure 4.2

6.5 User Interface

Figure 6.4 shows the main screen of *ConAn PaDi*. The following three paragraphs explain this user interface.

The list of all the patterns (1) shows their names followed by three numbers. The first black number shows from which order o they are. The green number indicates the number of elements the pattern contains and the last green number is the number of properties the pattern has. 'noname' means that the pattern could not be identified as to a pattern from the reference library. The list with the tuples (2) shows all the elements of the pattern. If exactly one tuple is highlighted, the structure of this tuple is shown the middle (5) as an UML [OBJE 99] diagram using ADvance [IC 01][BUCH 02]. In the list with the properties (3) the indexes are then replaced by the concrete class names. The name of the selected pattern (4) can be changed by the user.

The four top lists in the middle enable navigation through the patterns, using the pattern neighborhood. Clicking on an overloaded pattern (6), all the remaining tuples in the element list (2) are highlighted. Clicking on an almost pattern (7), the remaining properties (3) are highlighted. Behind the name of the almost pattern we see how many elements this pattern will gain by reducing some properties. Clicking on a sub pattern (8) highlights as well the

remaining properties (3). The last list is the link to the cover patterns (9).

The user can define filters to obtain a refined view, *e.g.* concentrate on a set of classes or hide some properties. The filters can be defined in the remaining lists. The filters have several criteria:

- *RejectClasses*
  If a tuple has any class of this collection, the tuple is rejected. Note: In the user interface the complement of this collection is shown as selected (10a).
  Example: If the class P is in the set of *rejectClasses*, then for order $o = 3$ the remaining tuples are: {C A B} and {Z X Y}. All the others ({C A P}, {Z X P}, {A P B}, {A P X} and {Y X P}) are rejected, because they have the class P in the tuple. In the user interface all tuples except P would be selected.

- *MustClasses*
  Classes which are in this collection must be in a tuple in order that the tuple pass the filter. (10b).
  Example: Is the class A in the set of *mustClasses*, then for order $o = 3$ the remaining tuples are: {C A P}, {C A B}, {A P B} and {A P X}. All the others ({Z X Y}, {Z X P} and {Y X P}) are rejected, because they do not have the class A in the tuple.

- *Orders*
  Only the selected orders are displayed (11).

- *Properties*
  Only properties of this collection pass the filter (12). In Figure 6.4 *isAbstract* is not marked, thus this property is not taken in consideration.

- *Methods*
  If a property is dependent on method, like *hasLocalDefinedMethod* $(X, m)_{lMeth}$ where *lMeth* is a method name, only properties with the selected methods are taken into account (13).

Importer

Before the patterns can be explored they have to be loaded in the repository. The *ConAn PaDi* Importer (Figure 6.5) provides this functionality. It needs as prerequisite a loaded *Moose* meta-model of the source code to analyze.

The desired properties can be selected and a first preselection based on class names can be made. Classes can be rejected or selected giving a string pattern like “PackageX*”. The order maximum can be entered as well.

6.6 Tool Architecture

This section gives a short overview of the tool architecture. The most important steps are shown in Figure 6.6. Grey dashed arrows mean information flow, whereas black solid

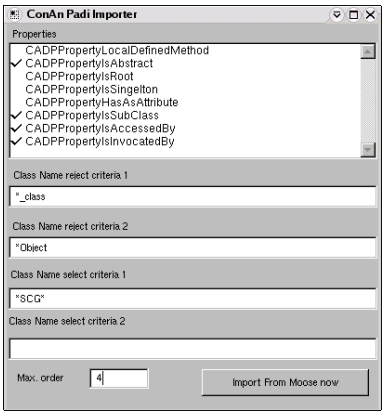


Figure 6.5: Pattern Browser: Import tool.

arrows mean program execution flow.

The tool has three main parts:

- The **Import Tool** imports the meta information from *Moose*. This is the first line in the Figure 6.6.
- The **Calculation** detects the patterns with FCA.
- The **Post process** filters the patterns with the desired criteria. This part removes and merges as well certain patterns, assigns their neighborhood and guesses their name.

This paragraph explains the 18 different steps and indicates where in this work the underlying theory is explained. The meta-model from the source code (1) is generated with *Moose* (2). A first filter (3) takes out the unwanted classes like meta classes or base classes which are not interesting. The classes are stored in the `CADPClassObject` wrapper in the `CADPClassRepository` (4). Each property definition is a subclass of `CADPProperty`, those subclasses are therefore the property definition repository (5).
Now the core calculation process can start: For each order the elements are built (6), more details are explained in Section 4.3.1. The properties have to be rebuilt for each order as well (7), read more in Section 4.3.2. These elements and properties are given to the *ConAn Black Box* (8). The incidence relation table $\mathcal{I}$ is defined (9). Now ConAn calculates the concepts (10) as explained in the Chapter 3.2.
The concepts are stored as patterns in the pattern repository (11). In this repository the patterns are grouped by their order.
All the steps so far are done once and they build up the patterns. The following steps are repeated each time the user changes the filter.
Applying the filter (12) (read Section 6.5) removes certain properties and elements from a pattern. Therefore patterns with empty tuple collection and/or empty property collection have to be removed (13). Reducing the properties may result an *unconnected* pattern (read

Section 4.5.2) which has to be removed (14). Some of the patterns may not be equivalent (read Section 4.5.3) and have to be merged (15). The last two steps assign the neighborhood and correcting the extent of the patterns (16) (read Section 4.6) and guesses, if possible, a name (17) (read Section 4.5.4). The results are stored in the repository for filtered patterns (18).

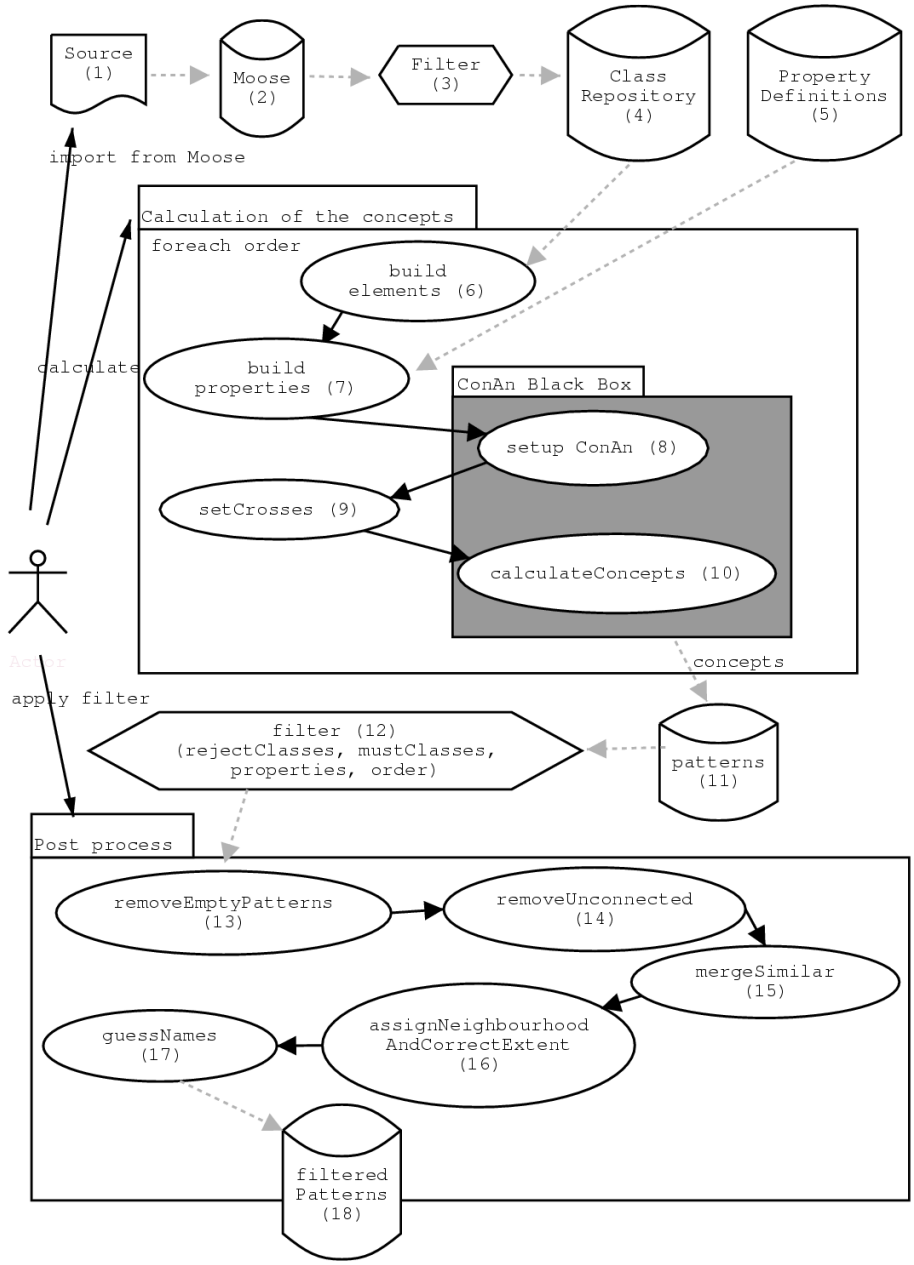


Figure 6.6: Pattern Browser: Tool architecture.

Chapter 7

Related Work

Our research is related to two major research areas. Namely, FCA applied to software engineering and pattern detection. The first we already discussed in Chapter 3. In this chapter we discuss the related work on pattern detection.

The following list gives an overview of the six publications which are as well compared in the Table 7.1.

- The work most related to our approach is done by Tonella and Antoniol. We use exactly the same approach as they proposed. They apply it to different C++ applications and extract a set of structural design patterns. Those patterns were enriched with non structural information about class members and method invocations [TONE 99]. This work is primarily based on their ideas, the additional contributions are:

  - Tonella and Antoniol do not evaluate the information from the lattice. They present the found patterns in isolation. Due to the concession for faster computation, like inductive element construction (Section 4.3.1), the relation information cannot taken directly from the lattice. We define the pattern neighborhood (Section 4.6) and show how it can be calculated.
  - To make the algorithm faster we propose an improvement for the algorithm of [TONE 99]: We eliminate all the combinations of a tuple and just take one representative for each permutation (Section 4.3.1).
  - We propose to take the information from a language independent meta-model instead from the source code itself. This makes the approach more powerful because it can be applied on applications to different programming languages (Section 6.2).
  - We propose to clearly separate the calculation of the concepts from the analyze process. This allows one to run just once the time consuming calculation of the concepts and store them in a repository. To analyze them, the user can apply different filter (Section 6.5) to this repository and explore the results.

  - With the *Fish Eye View*, we show a way to present the results in an clearly arranged way. We adapted the *Fish Eye Viewer* from ConAn to this special domain (Section 6.4).
  - To get a faster overview, we propose that in the post process the found patterns are compared with a reference library of well known (design) patterns (Section 4.5.4).
  - All those ideas are implemented in the tool *ConAn PaDi* (Chapter 6).
  - Last but not least, we clearly show how this approach is useful to software engineering (Chapter 2).

  This is the only approach besides ours which does not need a reference library.

- Brown presents in his master thesis a tool to detect design patterns in Smalltalk environments. He explains how to deal with the typeless language Smalltalk. The detection itself is then based on a cycle-detection of Corman [CORM 90]. The pattern can not be described in a higher abstraction [BROW 96]. Brown does not show a clearly generalizable approach to detect patterns. In his tool he has to write for each pattern a detection algorithm.

- Seemann and von Gudenberg use a compiler to generate graphs from the source code. This graph acts as the starting graph of a graph grammar that describes the design recovery process. The validation is made on popular patterns such as Composite and Strategy in the Java AWT package [SEEM 98].

- Keller et al. present an environment for the reverse engineering of design components based on the structural descriptions of design patterns. The validation is made with SPOOL on three large-scale C++ software systems. They store the meta-model as UML/CDIF and query then this model for patterns [KELL 99]. One year earlier Schauer and Keller already presented a prototype of this approach and showed the need of visualization of the found results [SCHA 98].

- Niere et al. provide a method and a corresponding tool which assist in design recovery and program understanding by recognizing instances of design patterns semi-automatically. The algorithm works incrementally and needs the domain and context knowledge given by a reverse engineer. To detect the patterns they use a special form of annotated abstract syntax graph (ASG). Using a subgraph matching algorithm allows them as well to define a pattern neighborhood as we gain out of the lattice (Section 4.6). An evaluation of the approach is made with the Java AWT and JGL libraries. [NIER 02].

- In Krämer and Prechtel approach, the patterns are stored as Prolog rules. Their tool Pat takes the meta-information directly from the C++ header files and queries them. The validation on the C++ libraries shows that the precision is around 40 percent [KRAM 96].

The following seven publications are somehow related but are not so close as the above mentioned.

- Baniassad and Murphy define conceptual modules as “a set of lines of source code that are to be treated as a logical unit.” [BANI 98] The difference between these techniques and pattern detection is the level of abstraction. Whereas conceptual modules represent only small algorithms or data structures, patterns illustrate the complex relationships among the large pieces of software and, equally important, embody informal explication of the rationale behind the suggested designs [KELL 99].

- Shull, Melo and Basili developed a manually inductive method to help discover design patterns in existing object-oriented software systems [SHUL 96]. We think that for big legacy systems computer aided detection is needed, otherwise the engineer loses too much time.

- Hanrandi and Ning introduce program analysis based on an *Event Base* and a *Plan Base*. Rudimentary events are constructed from source code. Plans are used to define the correlation between one or more events and they fire a new event which corresponds to the intention of the plan. Plans using events fired from other events gives a similar effect as the pattern neighborhood described in this work in Section 4.6 [HANR 90].

- Paul and Prakash have a similar approach as [HANR 90]. They use a matching algorithm for syntactic patterns based on non-deterministic finite automata [PAUL 94].

- Zündorf presents a graph pattern-matching tool in the PROGRES environment. PROGRES is a very high level multi-paradigm language for the specification of complex structured data types and their operations. Patterns are defined as graph transformation rules [ZÜND 96].

- Radermacher uses the same approach as [ZÜND 96]. He describes methodology and tool support. DiTo is their concrete implementation [RADE 99].

- Wills recognizes cliché, *i.e.*, commonly used computational structures, with the GRASPR system. Legacy code to be examined is represented as flow graphs by GRASPR, cliché are encoded as attributed graph grammar [WILL 94]. A cliché represents either a single algorithm, an algorithm fragment, or an ADT. These pieces of information are the basic building blocks of objects, from which patterns are created. While a cliché may represent the implementation of a piece of an object, it says nothing about the interaction of the object with other objects, which is the *raison d’être* of design patterns [BROW 96].

	Language Compatibility	meta-model	Detection approach	Reference Library	Tool (Programming language)	Validation	Performance
[TONE 99]	C++	source code itself	FCA	not needed	*not reported* (C++)	1 C++ application	within minutes
ConAn PaDi	independ	*Moose*	FCA	not needed (reference patterns for naming assigned by graph matching)	*ConAn PaDi* (Smalltalk)	3 Smalltalk applications	several hours
[BROW 96]	Smalltalk	source code itself	cycle-detection	hard-coded method	KT (Smalltalk)	4 Smalltalk applications	*not reported*
[SEEM 98]	Java	source code graph	graph grammar	reference graphs	*not reported* (Java)	Java AWT	*not reported*
[KELL 99]	independ	SPOOL	Poet 5.1	UML 1.1 / Poet 5.1	SPOOL (Java)	3 large C++ systems	*not reported*
[NIER 02]	Java	FUJABA	subgraph matching	ASG	FUJABA extension (Java)	2 Java libraries	within a minute
[KRAM 96]	C++	C++ header files	PROLOG queries	PROLOG rules	Pat (C++)	4 C++ applications	less than a minute

Table 7.1: Overview of the different detection approaches

Chapter 8

Conclusion

8.1 Summary

In this work we presented an approach to detect Software Patterns with Formal Concept Analysis (FCA). We showed how adequate such a detection is for the widely accepted claims of reverse engineering: Detecting Software Patterns can be beneficial for redocumentation and design recovery. We explained the mathematical background of FCA, where the basic definitions were defined and the used algorithm for calculating the concepts and the lattice were introduced. A short overview of the already explored fields of FCA applied in software engineering is given.

We described the approach step by step: Tuples of classes and the relations inside such a tuple serve as formal context. To avoid the combinatorial explosion of the elements, we use an incremental tuple construction. This context is handed over to the *ConAn* framework which calculates the concepts. The resulting concepts are post-processed to get better results. This post-filter primarily eliminates unwanted side-effects caused by how elements and properties are built. The found patterns are compared with a reference library of well-known patterns and become the name of the matched pattern. The patterns are set in relation. This means that neighborhood of a pattern is elaborated. A neighborhood pattern is *e.g.* a pattern which are almost like another one, but is missing some properties or has some properties more.

We presented our tool *ConAn PaDi*. This tool takes the information about the relations from the *Moose* meta-model. The *ConAn* framework is used for calculating the concepts. With *ConAn PaDi* the user can navigate through the found patterns. Besides the sequential list of all the patterns, the links from the pattern neighborhood can be used to navigate through the patterns. We successfully applied *ConAn PaDi* to three mid-sized Smalltalk applications. We have showed that we could reach our concrete objectives for reverse engineering. The biggest challenge turned out to be the immense calculation time needed to calculate pattern of higher order. The detection was as well limited to structural patterns, because with the used properties we could not infer behavioral patterns.

We compared our approach with other approaches and have seen that it is the only one which can infer patterns without a predefined pattern library.

8.2 Lessons Learned

- *Less is more.* This wisdom is true here as well. It is better to concentrate on few important properties: The more properties we have, the more patterns we find. It is true that more patterns are found, but all with just a few elements (one or two). We have too much statistical ”noise”. There is another aspect of working with too many properties: The calculation capacity of today’s computer is sooner reached.
- The time complexity of the used algorithm is $O(n^3)$, where n is the number of found concepts. We have seen that already in mid-size applications orders higher than four are too complex for our home PC. Its not enough to wait for computers which are twice or triple as fast. There is a need for good heuristics.
- The results can only be as good as the information from the meta-model is. When the input for FCA is already not reliable, the percentage of the fake candidates is even higher.

8.3 Future Work

- *Enhance the model with information on a higher abstraction level.*
  Instead of only using the structural information, we could use properties of an higher abstraction level. Such higher information could be properties like: isLeaf, isComponent, isFacade. Among the resulting patterns, behavioral patterns might be inferred.

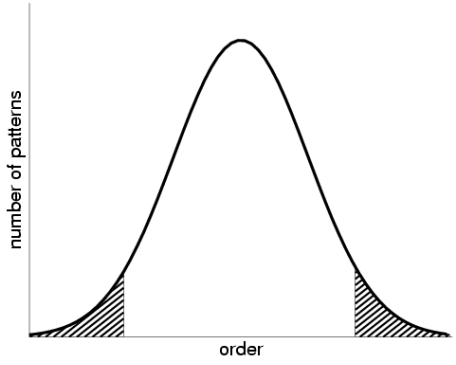


Figure 8.1: Unproblematic orders for calculation

- *Solve the scalability problem.*
  In an industrial environment *ConAn PaDi* has to be much faster. Results should be

available in real time or at least within seconds, otherwise the developer will not use this tool. One idea to enhance the speed performance is that not all orders are taken in consideration. Figure 8.1 shows approximately the expected found patterns in corelation with the order. We suppose that we just calculate the patterns before and after the peak of the curve, the hatched are in Figure 8.1. We expect that the patterns for the lower order are interesting to analyze the dependencies and the bigger on are interesting to understand the roles and gain an overview.

- *Better name guessing.*
  Use a better reference library to detect well-known patterns and improve the matching algorithm for them, *i.e.* making the matching algorithm more fuzzy.

Appendix A

Catalog of Properties

This chapter gives an overview of all the used properties in this work. A property is either a relationship inside the class tuple (*binary relation*) or a relation of a single class in the tuple (*unary relation*).

Notation

Name	***Name of the property***
Signature	*A short signature used in text, tables and figures.*
Reliability	*Not all of the relations are sure, some of them are just assumptions. This field indicates this reliability.*
Proof	*What is shown to the user as kind of proof why this property is assumed. Mainly with those properties where the reliability is not high, the user might be interested on the basis of which source code fragments the meta-model operator thinks that this relation is true.*
How the property is elaborated.	

A.1 Binary Relations R_B

Binary relations are the most important properties, because they *connect* classes. Just "fully connected" class tuples are taking into account. Unconnected tuples are not interesting, because they are not a pattern of its order (read more about this in Section 4.5.2).

Name	**isSubclass**
Signature	$(X, Y)_{Sub}$
Reliability	absolute
Proof	Class definition of X
isSubclass is true, when X inherits from Y.	

Name	**accesses**
Signature	$(X, Y)_{Acc}$
Reliability	absolute
Proof	Method body
accesses is true, when at least one entity in a method of the class Y sends a message to the class side of a class X.	

Name	**hasAsAttribute**
Signature	$(X, Y)_{Att}$
Reliability	medium
Proof	All methods of X which read / write this instance variable of the type Y
hasAsAttribute is true, when ...	

Name	**invocates**
Signature	$(X, Y)_{Inv}$
Reliability	low
Proof	All methods of Y which have an invocation to X
invocates is true, when in a method of Y an entity is used which could have the interface of class X.	

Name	**usesLocally**
Signature	$(X, Y)_{lUse}$
Reliability	medium
Proof	All methods of Y which locally use X
usesLocally is true, when in a method of Y an entity is locally used which could have the interface of class X.	

A.2 Unary Relations R_U

Name	**isAbstract**
Signature	$(X)_{Abstr}$
Reliability	high
Proof	All abstract methods
isAbstract is true when at least one method is abstract.	

Name	**isRoot**
Signature	$(X)_{Root}$
Reliability	absolute
Proof	Class definition
isRoot is true when the class X inherits from `Object` or `nil`.	

Name	**isSingelton**
Signature	$(X)_{Singl}$
Reliability	medium
Proof	method called uniqueInstance
isSingelton is true, when the class X has a method called *uniqueInstance* on the class side.	

Name	**hasLocalDefinedMethod**
Signature	$(X, m)_{lMeth}$
Reliability	absolute
Proof	method definition m of class X
hasLocalDefinedMethod is true, when the class X has a method called m on the instance side.	

Bibliography

[IC 01] IC & C GmBH Software Foundations, Papenhoehe 14, D-25335 Elmshorn/Hamburg, Germany. *ADvance User's Guide*, August 2001. (p 45)

[AEBI 03] T. Aebi. Extracting Architectural Information using different Levels of Collaboration. Master thesis, University of Bern, September 2003. (pp 32, 36)

[ANDE 97] U. Andelfinger. Diskursive Anforderungsanalyse. Ein Beitrag zum Reduktionsproblem bei Systementwicklungen in der Informatik. Peter Lang, Frankfurt, 1997. (p 19)

[APPL] B. Appleton. *Patterns and software: Essential concepts and terminology.* http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html. (p 1)

[ARÉV 03a] G. Arévalo. *Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis*. In LMO 03: Langages et Modeles à Objets. Hermes, Paris, 2003. (p 20)

[ARÉV 03b] G. Arévalo. *X-Ray Views on a Class using Concept Analysis*. In Proceedings of WOOR 2003 (4th International Workshop on Object-Oriented Reengineering), pages 76–80. University of Antwerp, 2003. (p 20)

[BALL 99] T. Ball. *The Concept of Dynamic Analysis*. In Proceedings of ESEC/FSE '99, number 1687 in LNCS, pages 216–234, 1999. (p 20)

[BANI 98] E. L. A. Baniassad and G. C. Murphy. *Conceptual module querying for software reengineering*. In Proceedings of the 20th international conference on Software engineering, pages 64–73. IEEE Computer Society, 1998. (p 52)

[BECK 94] K. Beck and R. Johnson. *Patterns Generate Architectures*. In M. Tokoro and R. Pareschi, editors, Proceedings ECOOP '94, volume 821 of *LNCS*, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag. (pp 7, 8)

[BIRK 40] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1940. (p 12)

[BÖ 01] K. Böttger, R. Schwitter, D. Richards, O. Aguilera, and D. Mollá. *Reconciling Use Cases via Controlled Language and Graphical Models*. In INAP'2001 - Proceedings of the 14th International Conference on Applications of Prolog, pages 20–22, Japan, October 2001. University of Tokyo. (p 19)

[BOJI 00] D. Bojic and D. Velasevic. *Reverse Engineering of Use Case Realizations in UML*. In SAC2000. ACM, 2000. (p 20)

[BROW 96] K. Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters thesis, North Carolina State University, 1996. (pp 51, 52, 53)

[BUCH 02] F. Buchli. *An explicit model for ADVance*. Informatikprojekt, University of Bern, December 2002. (p 45)

[CANF 99] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. *A Case Study of Applying an Eclectic Approach to Identify Objects in Code*. In Workshop on Program Comprehension, pages 136–143. IEEE, 1999. (p 20)

[CASA 98] E. Casais. *Re-Engineering Object-Oriented Legacy Systems*. Journal of Object-Oriented Programming, vol. 10, no. 8, pages 45–52, January 1998. (p 4)

[CHIK 90] E. J. Chikofsky and J. H. Cross, II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, pages 13–17, January 1990. (pp 1, 5, 6, 35)

[CORM 90] T. H. Corman, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, 1990. (p 51)

[DEME 99] S. Demeyer, S. Ducasse, and M. Lanza. *A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization*. In F. Balmas, M. Blaha, and S. Rugaber, editors, Proceedings WCRE '99 (6th Working Conference on Reverse Engineering). IEEE, October 1999. (p 33)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 – The FAMOOS Information Exchange Model*. Research report, University of Bern, 2001. (p 40)

[DÜ 98] S. Düwel and W. Hesse. *Identifying Candidate Objects During System Analysis*. In Proceedings of CAiSE'98/IFIP 8.1 3rd International Workshop on Evaluation of Modelling Methods in System Analysis and Design (EMMSAD'98), Pisa, 1998. (p 19)

[DÜ 99] S. Düwel. *Enhancing System Analysis by Means of Formal Concept Analysis*. In Conference on Advanced Information Systems Engineering 6th Doctoral Consortium, Heidelberg, Germany, June 1999. (p 19)

[DÜ 00] S. Düwel and W. Hesse. *Bridging the gap between Use Case Analysis and Class Structure Design by Formal Concept Analysis*. In J. Ebert and U. Frank, editors, Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proceedings "Modellierung 2000", pages 27–40, Koblenz, 2000. Fölbach-Verlag. (p 19)

[DUCA 00] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000. (p 40)

[DUCA 01a] S. Ducasse, M. Lanza, and S. Tichelaar. *The Moose Reengineering Environment*. Smalltalk Chronicles, August 2001. (p 40)

[DUCA 01b] S. Ducasse and M. Lanza. *Towards a Methodology for the Understanding of Object-Oriented Systems*. Technique et science informatiques, vol. 20, no. 4, pages 539–566, 2001. (p 33)

[EISE 01a] T. Eisenbarth, R. Koschke, and D. Simon. *Aiding Program Comprehension by Static and Dynamic Feature Analysis*. In Proceedings ICSM. IEEE Computer Society Press, 2001. (p 20)

[EISE 01b] T. Eisenbarth, R. Koschke, and D. Simon. *Feature-Driven Program Understanding Using Concept Analysis of Execution Traces*. In 9th International Workshop on Program Comprehension, pages 300–309. IEEE, 2001. (p 20)

[FISC 98] B. Fischer. *Specification-based Browsing of Software Component Libraries*. In Automated Software Engineering, pages 74–83, 1998. (p 19)

[FUNK 95] P. Funk, A. Lewien, and G. Snelting. *Algorithms for Concept Lattice Decomposition and their Application*. Research Report 95-09, Computer Science Dept., Technische Universitat Braunschweig, 1995. (p 20)

[FURN 81] G. W. Furnas. *The Fisheye View: A New Look at Structured Files*. Research Report #81-11221-9, Murray Hill, New Jersey 07974, U.S.A., 12 1981. (p 43)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass., 1995. (pp 1, 7, 34)

[GANT 99] B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer Verlag, 1999. (pp 1, 10, 11, 15, 16, 25, 43)

[GODI 93] R. Godin and H. Mili. *Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices*. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, pages 394–410, October 1993. (p 20)

[GODI 98] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. *Design of class hierarchies based on concept (Galois) lattices*. Theory and Application of Object Systems, vol. 4, no. 2, pages 117–134, 1998. (pp 11, 15)

[HANR 90] M. Hanrandi and J. Ning. *Knowledge-Based Program Analysis*. IEEE Transaction on Software Engineering, vol. 7, no. 1, pages 74–81, 1990. (p 52)

[HUCH 99] M. Huchard and H. Leblanc. *From Java classes to Java interfaces through Galois lattices*. In Actes de ORDAL'99: 3rd International Conference on Orders, Algorithms and Applications, pages 211–216, Montpellier, 1999. (p 20)

[JOHN 98] D. S. Johnson and M. Yannakakis. *On Generating all Maximal Indepndent Sets*. Information Processing Letters, vol. 27, pages 119–123, 1998. (p 31)

[KELL 99] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. *Pattern-Based Reverse Engineering of Design Components*. In Proceedings of ICSE '99, pages 226–235. IEEE Computer Society Press, May 1999. (pp 9, 51, 52, 53)

[KRAM 96] C. Kramer and L. Prechelt. *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*. In Proceedingsof WCRE '96. IEEE, November 1996. (pp 9, 51, 53)

[KRON 94] M. Krone and G. Snelting. *On The Inference of Configuration Structures from Source Code*. In Proceedings of ICSE 1994, pages 49–57, 1994. (p 20)

[KUIP 00] T. Kuipers and L. Moonen. *Types and Concept Analysis for Legacy Systems*. Research Report SEN-R0017, Centrum voor Wiskunde en Informatica, July 2000. (p 20)

[KUZN 01] S. Kuznetsov and S. Obëdkov. *Comparing Performance of Algorithms for Generating Concept Lattices*. In Proc. Int. Workshop on Concept Lattices-based KDD, 2001. (pp 25, 31)

[LAMP 95] J. Lamping, R. Rao, and P. Pirolli. *A Focus + Context Technique Based on Hyperbolic Geometry for Visualising Larges Hierarchies*. In Proceedings of CHI '95, 1995. (p 43)

[LANZ 99] M. Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma thesis, University of Bern, October 1999. (pp 5, 33, 43)

[LANZ 01] M. Lanza, S. Ducasse, and L. Steiger. *Understanding Software Evolution using a Flexible Query Engine*. In Proceedings of the Workshop on Formal Foundations of Software Evolution, 2001. (p 33)

[LANZ 03] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003. (pp 4, 5, 6, 33)

[LEHM 85] M. M. Lehman and L. Belady. Program Evolution - Processes of Software Change. London Academic Press, 1985. (pp 1, 4)

[LIND 95] C. Lindig. *Concept-Based Component Retrieval*. In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, editors, Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, pages 21–25, August 1995. (p 19)

[NIER 02] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. *Towards pattern-based design recovery*. In Proceedings of the 24th international conference on Software engineering, pages 338–348. ACM Press, 2002. (pp 1, 9, 51, 53)

[OBJE 99] Object Management Group. *Unified Modeling Language (version 1.3)*. Research report, Object Management Group, June 1999. (p 45)

[PAUL 94] S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns*. IEEE Transactions on Software Engineering, vol. 20, no. 6, pages 463–475, jun 1994. (p 52)

[RADE 99] A. Radermacher. *Support for Design Patterns Through Graph Transformation Tools*. In AGTIVE, pages 111–126, 1999. (p 52)

[RICH 02a] D. Richards and K. Boettger. *Assisting Decision Making in Requirements Reconciliation*. In Proceedings of the 7th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2002), Brazil, September 2002. (p 19)

[RICH 02b] D. Richards and K. Boettger. *A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices*. In Proceedings of 15th Australian Joint Conference on Artificial Intelligence, 2002. (p 19)

[RICH 02c] D. Richards, K. Boettger, and A. Fure. *RECOCASE-tool: A CASE tool for RECOnciling Requirements Viewpoints*. In Proceedings of the 7th Australian Workshop on Requirements Engineering, AWRE'2002, 2002. (p 19)

[RICH 02d] D. Richards and K. Boettger. *Representing Requirements in Natural Language as Concept Lattices*. In 22nd Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGES), (ES2002), Cambridge, December 2002. (p 19)

[RICH 02e] D. Richards, K. Boettger, and A. Fure. *Using RECOCASE to Compare Use Cases from Multiple Viewpoints*. In Proceedings of the 13th Australasian Conference on Information Systems ACIS 2002, Melbourne, December 2002. (p 19)

[RUGA 98] S. Rugaber and J. White. *Restoring a Legacy: Lessons Learned*. IEEE Software, vol. 15, no. 4, pages 28–33, July 1998. (p 4)

[SAHR 97] H. Sahraoui, W. Melo, H. Lounis, and F. Dumont. *Applying concept formation methods to object identification in procedural code*. In Proceedings of International Conference on Automated Software Engineering (ASE '97), pages 210–218. IEEE, November 1997. (p 20)

[SCHA 98] R. Schauer and R. Keller. *Pattern Visualization for Software Comprehension*. In 6th International Workshop on Program Comprehension (Ischia, Italy), pages 4–12, 1998. (pp 9, 51)

[SCHM 95] D. C. Schmidt. *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. Communications of the ACM, vol. 38, no. 10, pages 65–74, October 1995. (p 7)

[SCHU 02] S. Schupp, M. Krishnamoorthy, M. Zalewski, and J. Kilbride. *The "Right" Level of Abstraction—Assessing Reusable Software with Formal Concept Analysis*. In G. Angelova, D. Corbett, and U. Priss, editors, Foundations and Applications of Conceptual Structures - Contributions to ICCS 2002, pages 74–91. Bulgarian Academy of Sciences, 2002. (p 20)

[SEEM 98] J. Seemann and J. W. von Gudenberg. *Pattern-based design recovery of Java software*. In Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, pages 10–16. ACM Press, 1998. (pp 9, 51, 53)

[SHUL 96] F. Shull, W. L. Melo, and V. R. Basili. *An inductive method for discovering design patterns from object-oriented software systems*. Research Report CS-TR-3597, University of Maryland Computer Science Department, 1996. (p 52)

[SIFF 97] M. Siff and T. Reps. *Identifying Modules via Concept Analysis*. In Proc. of. the Internation Conference on Software Maintenance, pages 170–179. IEEE Computer Society Press, 1997. (pp 14, 15, 20, 24, 43)

[SNEL 96] G. Snelting. *Reengineering of Configurations Based on Mathematical Concept Analysis*. ACM Transactions on Software Engineering and Methodology, vol. 5, no. 2, pages 146–189, April 1996. (p 20)

[SNEL 97] G. Snelting and F. Tip. *Reengineering Class Hierarchies Using Concept Analysis*. Research Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997. (p 20)

[SNEL 98a] G. Snelting and F. Tip. *Reengineering Class Hierarchies Using Concept Analysis*. In ACM Trans. Programming Languages and Systems, 1998. (pp 15, 20)

[SNEL 98b] G. Snelting. *Concept Analysis - A New Framework for Program Understanding*. In SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 1–10, Montreal, Canada, June 1998. (p 20)

[SPIV 89] J. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, 1989. (p 19)

[TICH 01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001. (pp 36, 40)

[TILL 03a] 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 the First International Conference on Formal Concept Analysis - ICFCA'03. Springer-Verlag, February 2003. (p 19)

[TILL 03b] T. Tilley. *Towards an FCA based tool for visualising Formal Specifications*. In B. Ganter and A. de Moor, editors, Using Conceptual Structures: Contributions to ICCS 2003, pages 227–240. Shaker Verlag, 2003. (p 19)

[TONE 99] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference*. In Proceedings ICSM '99, pages 230–238, October 1999. (pp 2, 7, 9, 20, 23, 24, 27, 36, 50, 53)

[TONE 01] P. Tonella. *Concept Analysis for Module Restructuring*. IEEE Trans on Software Engineering, vol. 27, no. 4, pages 351–363, April 2001. (p 20)

[VAN 99] A. van Deursen and T. Kuipers. *Identifying Objects using Cluster and Concept Analysis*. In Proceedings of 21st International Conference on Software Engineering, ICSE-99, pages 246–255. ACM, 1999. (p 20)

[WILL 81] R. Wille. *Restructuring lattice theory: An approach based on hierarchies of concepts*. Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute, vol. 83, pages 445–470, September 1981. (pp 10, 14)

[WILL 94] L. M. Wills. *Using Attributed Flow Graph Parsing to Recognize Programs*. In Int. Workshop on Graph Grammars and Their Application to Computer Science, Williamsburg, Virginia, November 1994. (p 52)

[ZÜND 96] A. Zündorf. *Graph pattern matching in PROGRES*. In Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci., volume 1073, pages 454–468. Springer, 1996. (p 52)