High-Level Views in Object-Oriented Systems using Formal Concept Analysis

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Gabriela Beatriz Arévalo
von Argentinien

Leiter der Arbeit:
Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

High-Level Views in Object-Oriented Systems using Formal Concept Analysis

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Gabriela Beatriz Arévalo
von Argentinien

Leiter der Arbeit:
Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 14.12.2004

Der Dekan:
Prof. Paul Messerli

This work is dedicated
to Mom and Dad

Abstract

Within object-oriented systems there are different meaningful dependencies between different objects. These dependencies reveal *contracts*, *collaborations* and *relationships* between classes, methods, packages and any development unit in the systems. In most of the cases, these dependencies are not explicit in the code. This problem is due to inadequate or out-of-date documentation and mechanisms such as dynamic binding, inheritance and polymorphism that obscure the presence of existing dependencies.

These dependencies play an important part in implicit contracts between the various software artifacts of the system. It is therefore essential that a developer, who has to make changes or extensions to an object-oriented system, understands the dependencies among the classes. Lack of understanding increases the risk that seemingly innocuous changes break the implicit existing contracts in the system. In short, implicit, undocumented dependencies lead to *fragile systems* that are difficult to extend or modify correctly.

In this thesis we develop an approach – based on a methodology and a tool support – to recover this *implicit* information and generate *high-level views* of a system at different abstraction levels, using a formal clustering technique called Formal Concept Analysis (FCA). With these *views*, we help to build the first mental model of a system. Thus the implicit or lost information is made explicit and we are able to find uses of coding styles, possible bottlenecks and weakpoints of a system, identify eventual contracts between the entities, *patterns* based on the dependencies and – if possible – propose possible solutions to correct problems in the code. With this approach we also evaluate which are the advantages and disadvantages of using a clustering technique in software reverse engineering.

Acknowledgements

This adventure started in October 2000 when I arrived in Bern to work in the Software Composition Group. And the 4 years I have spent since this date, I must confess has been an interesting experience.

After all these four years, I have got a list of people who contributed in different ways to my Ph.D. thesis, for which I would like to express thanks.

I thank Prof. Oscar Nierstrasz. He believed in my ability to do the Ph.D. thesis in the Software Composition Group, he gave me guidance and support throughout these four years, and he offered me useful professional advice.

I thank Prof. Stéphane Ducasse. He is the "motivation machine" of the Software Composition Group and a friend that I met in SCG. He helped me from the beginning of the thesis with his useful discussions, guidance and encouragement in the university and all the trips (to conferences) we have made together. From the personal viewpoint, I thank him and all his family: Florence, Quentin and Thibaut that were like my family, especially in the first years of my stay in Switzerland. I enjoyed every moment I spent with them.

I thank Prof. Marianne Huchard and Prof. Giuliano Antoniol. They accepted to be my external reviewers of this thesis, and offered useful comments on my thesis. They also accepted to come to my PhD defense in spite of their busy agendas, and I appreciate that a lot.

I thank Prof. Horst Bunke. for accepting to chair my Ph.D. defense.

I thank Tamar Richner, Juan Carlos Cruz and Sander Tichelaar. Their friendship and their friendly advice have been very important and have encouraged me a lot during these 4 years.

I thank Michele Lanza and Orla Greevy. They were my office colleagues and shared with me the room 106 every day of these 4 years. I appreciate them a lot because they were the proof-readers of the first drafts of my thesis with Tudor Gîrba. We laughed a lot, we had crazy discussions and supported ourselves in good and bad moments.

I thank Franch Buchli. He was my master student and helped me to improve my methodology and my tool. I appreciate all the work and effort he made for ConAn.

I thank all the rest of people in the SCG: Franz Achermann, Alexandre Bergel, Calogero Butera, Markus Denker, Markus Gaelli, Tudor Gîrba, Nathanael Schaerli, Laura Ponisio, Daniele Talerico, David Vogel, Roel Wuyts. We shared nice barbecues, tea-times, coffee-times and interesting discussions in the University.

I thank Therese Schmid. She was my main support in all the administrative papers in the University, but also one of my providers of chocolates and cookies in the SCG.

I thank all people I met in the conferences. I have had nice moments with them and also useful discussions to help me to improve my work.

I thank all people I met in "Cruz del Sur" and Misión Católica in Bern. They represent my "other life" during my stay in Switzerland, especially during weekends

I thank my friends Ana Maria Hintermann, Silvana Nicolini, Marcelo and Silvia Morard, Trudi Crego and Alejandro Strickler, Karina Mayocchi and Alejandro Juroczko. We spent crazy moments together and were

the supporters of this thesis too.

I thank my brothers and sisters Dario, Cynthia, Patricia and Cristian. My *for ever* fan club and the supporters that any team in the world would like to have.

I thank Jorge, who encouraged me during last year with his love and his advice.

I thank Papá and Mamá. They supported me with all their strengths and love in this project, although I was so far from home. If God gave me the opportunity to choose another parents, I would choose them again. I love them !

Thank you everybody from the depth of my heart !!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Today large organizations are not only faced with the problem of replacing their information systems with completely new ones, but they have to maintain or gain control over their legacy systems [DDN02]. During the maintenance phase of the software lifecycle, developers must constantly cope with evolving requirements, such as new platforms, new technologies, new users' needs or new functionalities. These changes are inevitable in the software lifecycle [Par94].

When the developer must face with these changes, the first step is reverse engineering the system. According to Chikofsky's definition [CC92] *"reverse engineering is the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction"*. The main goal in this step of a reengineering process of a software is to generate a mental model of the system [SFM99]. This mental model must be the first step in analyzing a software. In any software this mental model is the first fingerprints of the system identifying components and relationships between them at different abstraction levels. Specifically, in object-oriented systems, the components can be methods, variables, classes, or a set of classes, the interrelationships can be inheritance, methods calls, etc, and the abstraction levels identified then are class-, class hierarchy- or application-levels.

Unfortunately building this mental model is not a trivial task because not all the relationships are explicit at the source code level. Besides that if we want to detect them using any kind of documentation – such as manuals about design in case of industrial projects or simple code comments –, most of the cases it is out-of-date or insufficient. These relationships are important because they reveal meaningful dependencies between different components of the system. When this knowledge is not explicit or is lost, any change in the system is complex and can break the current functionalities or introduce new unexpected relationships increasing the complexity of the system to be understood.

Summarizing from these issues, we state that the main problem is:

> Without precise and updated documentation, a system is like a *puzzle*, where the pieces have no order at all. Thus, the software engineer is not able to infer what is implemented, what are the imposed constraints and how the system is working. The software engineer needs an approach to cope with this problem to be able to analyze a system.

In this thesis we develop an approach where we recover this *implicit* information and generate *high-level views* of a system using a formal clustering technique called Formal Concept Analysis (FCA) [GW99]. With these *views*, we help to build the first mental model of a system. Thus the implicit or lost information is made explicit and we are able to find uses of coding styles, possible bottlenecks and weakpoints of a system, identify eventual contracts [SLMD96] between the entities, *patterns* based on the dependencies and – if possible – propose possible solutions to correct problems in the code.

1.1 The Problem

In Introduction we state that the main problem in building a mental model is that the system has hidden information that must be explicit to understand how the system is working.

To cope with this problem, this thesis is focused on two research directions based on the following questions:

- How we can identify and detect implicit dependencies between the objects in a system?
- What is the adequate tool support to provide an infrastructure to the dependencies detection ?

Let's analyze in detail which are the different problems in these two research directions.

Identification of Dependencies: In object-oriented systems there are different meaningful dependencies between different objects. These dependencies reveal contracts, collaborations and relationships between classes, methods, packages and any development unit in the system. These dependencies are determined by different building mechanisms of object-oriented systems: *reuse, delegation, data encapsulation, dynamic binding, inheritance* and *polymorphism*. Although these mechanisms represent advantages of object-oriented systems, the maintainers of object-oriented systems must cope with several problems. The most important is that not all these dependencies are explicit in the system. Some examples are following:

- *Example 1:* In Smalltalk, there exists the `Collection` class hierarchy that defines all the classes responsible for managing any kind of collections. Let's take only the subhierarchy whose root is the OrderedCollection class. We also consider in this subhierarchy the SequeanceableCollection class which is the superclass of OrderedCollection.
  OrderedCollection has six subclasses defined in two inheritance levels. This class defines the method size that calculates the amount of elements contained in the collection, and the method is inherited (and not overriden) by six subclasses. If we modify the behavior of the method `size`, or if we remove it from the class, we are breaking the functionalities of the subclasses [MS98]. In case of removal, a bug appears because the method size in SequenceableCollection is abstract.
- *Example 2:* A design pattern (such as Composite Pattern [GHJV95]) is implemented in a system, and the developer did not use the right names to identify the components. Then we add a new method that determine a new collaboration between the three classes of the pattern. By adding this new method we are hiding the existence of the pattern in the code, and we are breaking the current functionalities of the pattern.
- *Example 3:* One of the salient features of Smalltalk is the fully reified compilation process. The developer may extend Smalltalk semantics providing new compile-time features by extending the classes hierarchies whose roots are Parser, ProgramNodeBuilder, ProgramNode, CodeStream, CompiledMethod, NameScope, Compiler, CompilerErrorHandler and Decompiler [Riv96]. If the developer is not aware of the relationships between all these class hierarchies, he can break the existing language semantics or create a wrong one.

The existence of these implicit dependencies is followed by undesirable characteristics such as a poorly structured source code, missing or incomplete design specifications, non-existing or out of date documentation, high level of redundancy or extremely complex modules. Several examples, like ones mentioned before, are enough to see that discovering these dependencies is important if we want to perform any change in the code. These dependencies have three main features:

- The dependencies are not constrained to a specific development unit. They appear at the class-, class hierarchy- or application-level of a system. *Example 1* is a case of dependencies introduced at the class hierarchy-level. *Examples 2* and *3* are cases of dependencies at the application-level
- The dependencies do not appear isolated in the system. *Example 1* shows us the use of inheritance and behavior reuse between classes.

- The dependencies appear as recurring situations in the system. For example, the case of hook methods [WBWW90] can appear several times in a class hierarchy.

We see that when a developer must reengineer a system, these problems make the task complex to manage because any kind of unstructured changes in the system can either break existing dependencies or duplicate information including existing dependencies among several objects.

Tool Support: Within the context of software tools, approaches such as Godin et. al. [GMM+95] and Dekel [Dek03] cope partially with these problems using also Formal Concept Analysis. But their main drawbacks are that they use explicit dependencies (such as inheritance or method calls), and they do not discover implicit dependencies. Besides that, the interpretation of the results is based on some knowledge about the mathematical background of FCA.

> In this thesis we propose an approach composed of a methodology and a tool support to detect implicit dependencies. Using FCA as a base tool, we build three different tools to analyze an object-oriented system at different abstraction levels: class-level, class-hierarchy level and application level. In each level, we provide *high-level views* that allow to discover which are the implicit information at each level, and analyze the application.

Our research is driven by the following questions:

- Do we understand how the system is implemented? What are the constraints or limits imposed in the system?
- How can we detect implicit unanticipated dependencies?
- How are the mechanisms such as polymorphism and inheritance used in the system?
- How can we discover defects introduced in the systems?
- How can we discover recurring situations (*patterns-like*) of dependencies in the systems?
- Is the mental model generated by *high-level views* meaningful enough (in terms of information) to understand the system?
- Which are the advantages and disadvantages of using a clustering technique such FCA as a metatool?

1.2 Our Approach: Formal Concept Analysis in Object-Oriented Systems

Formal Concept Analysis provides a formal framework for recognizing groups of elements sharing common properties. Based on FCA, our approach consists of a methodology and atool to support the dependencies detection. In the methodology, we characterize the software relationships as FCA properties and the software artifacts as FCA elements. The group of elements and properties (named as *concepts*) reveals existing implicit and explicit dependencies in the system. With these concepts we build *high-level views* on the code to get the first mental model of the system. These *high-level views* help the maintainer to discover the internal workings of a system, possible constraints and defects introduced in the system. These *high-level views* are defined in three different abstraction levels: *XRay views* at class level, *hierarchy schemas* at class hierarchy level and *collaboration patterns* at application level.

- **XRay Views on Classes:** Analyzing the state and behavior of a class, we generate XRay views to show us which are the different dependencies between methods and variables in a class.

- **Hierarchy Schemas:** Analyzing the state and behavior of all the classes in a hierarchy, we identify the different recurrent dependencies between the classes. They help the developer to understand which are the common and irregular design decisions when the hierarchy was built, and eventual refactoring cases (if any) that can be applied.
- **Collaboration Patterns:** Analyzing structural relationships between classes in a system, we discover *possible patterns* that appear in the system. With them, we analyze how the system was built and which were the main constraints imposed in it.

Based on a generic system, Figures 1.1, 1.2 and 1.3 show the ideas of grouping software artifacts to generate *XRay Views* at class level, *Hierarchy Schemas* at class hierarchy level and *Collaboration Patterns* at application level.

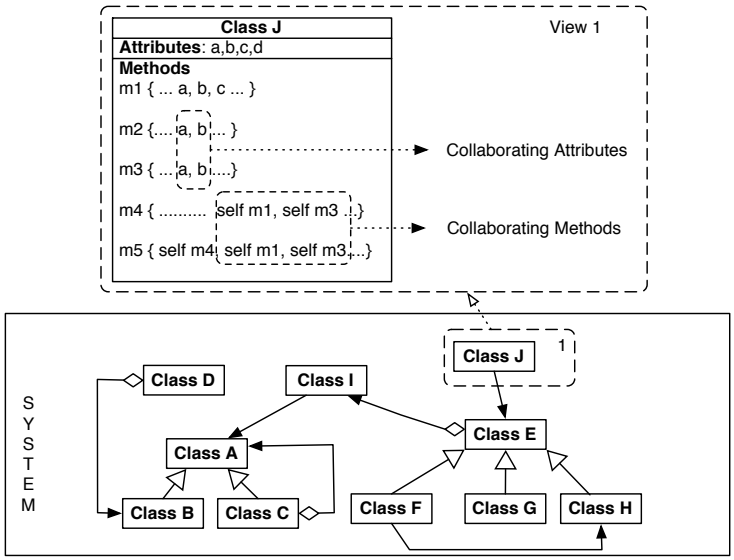

Figure 1.1: XRay views applied on a class

The three different high-level views are complementary. If we discover a collaboration pattern that involves sub-hierarchy, and we are interested in analyzing it, we generate the hierarchical schemas on that sub-hierarchy. Or if we discover a hierarchical schema that involves a *God* class, and we are interested in analyzing it, we generate the XRay views on the class. But this does not mean that a high-level view is included in another high-level view of a different abstraction level. For example, a XRay view will not appear in a hierarchy schema, and this is because the different high-level views are defined with different information in each abstraction level.

The tool support consists of a tool named ConAn implemented in VisualWorks [Vis03]. It is composed of 4 components: a *base framework* is the implementation of FCA basics (definition of elements, properties and incidence table, algorithms, and visualization and navigation capabilities of the lattice). The other 3 components are implemented on top of the base framework that allows the application of FCA and analysis of a system at three different abstraction levels.

1.3 Contributions

The main contributions of this thesis comprise:

- Introduction of three different high-level views based on FCA: *XRay Views* on classes, *Hierarchy Schemas* on class hierarchies and *Collaboration Patterns* on applications.

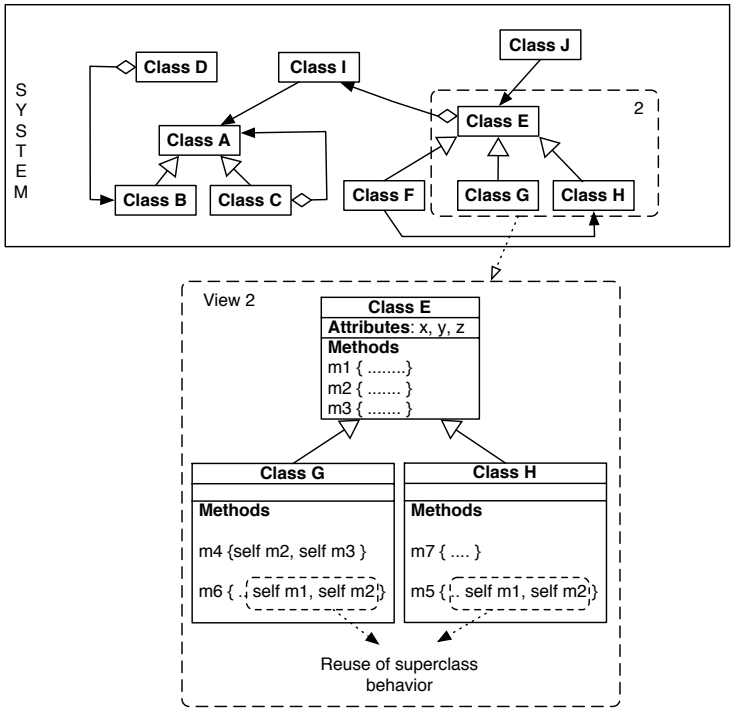

Figure 1.2: Hierarchy Schemas on a Class Hierarchy

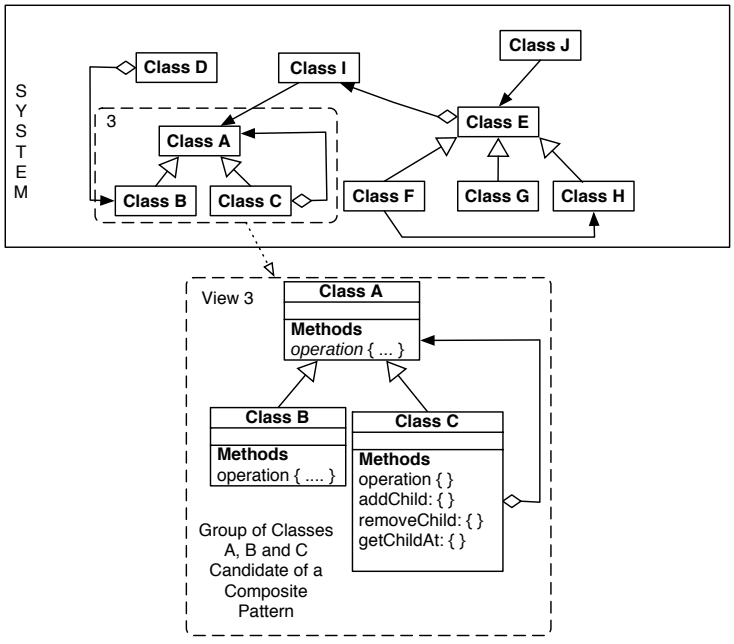

Figure 1.3: Collaboration Pattern identified in the application

- Development of a methodology to analyze object-oriented system using *Formal Concept Analysis*.
- Development of a tool framework called *ConAn* that allows us to define three different tools where we apply the methodology mentioned previously at class-, class hierarchy-, and application level.
- Analysis of the advantages and constraints of using Formal Concept Analysis in building high-level views to reverse engineer a system, and in building support tools to generate and analyze the high-level views.
- Interpretation of results of the high-level views without having any knowledge of mathematical background of FCA.

1.4 Thesis Outline

This thesis is outlined as follows:

- Chapter 2 introduces the concept of *implicit* dependencies and we identify the different problems we find in object-oriented code, and we explain why we need three different high-level views. We also show how the classical reverse engineering and clustering techniques cope with the different problems.
- Chapter 3 introduces in detail the approach based on Formal Concept Analysis identifying the different issues to take into account when using this approach at three different abstraction levels. We also summarize how Formal Concept Analysis is used in different reverse engineering problems.
- Chapter 4, 5 and 6 introduce the different high-level views we have developed: *XRay Views* applied on classes, *Hierarchy Schemas* applied on class hierarchies and *Collaboration Patterns* on applications. In the three cases, we explain in detail how FCA is applied in classes, class hierarchies and application respectively. Then we explain how the high-level views are generated based on the results provided by FCA, and how we interpret the information we have in the high-level views. All cases are validated with case studies and we also show those results. Finally, we analyze how the issues of the methodology –mentioned in general in the Chapter 3– affect the generation of the high-level views.
- Chapter 7 presents the lessons learned developing a FCA-based approach, conclusions and future work.
- Appendix A presents details about the tool framework named ConAn
- Appendix B is a complement to the thesis and explains in detail the mathematical background and algorithms of Formal Concept Analysis.

Chapter 2

Dependencies in Object-Oriented Systems

This thesis is about the application of a conceptual clustering technique called Formal Concept Analysis to generate *high-level views* to detect implicit contracts determined by different dependencies between objects in object-oriented systems. In this chapter, we explain the main context of our approach and which are the different problems that object-oriented systems present. We also show how existing reverse engineering and clustering approaches cope with these problems.

2.1 Introduction

All software systems[1] are exposed to changes during their lifecycle [Par94]. Any change in the systems implies an evolution at large or small scale. Manny Lehman and Les Belady [LB85] have identified two main laws of software evolution:

- *Law of continuing change:* A program that is used in a real-world environment *must* change, or become progressively less useful in that environment.
- *Law of increasing complexity:* As a program evolves, it becomes more *complex*, and extra resources are needed to preserve and simplify its structure.

From these two laws we conclude that the changes are inevitable [Par94] and are not free, the maintainer has to pay a price in terms of *complexity*.

When a system must be changed, it must follow a *reengineering* process. According to Chikofsky et. al. [CC92],

> *Reengineering* is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of a new form.

As stated by the definition, *Reengineering* consists of two main activities, namely the *examination* and the *alteration* of a subject system. More formal terms for these activities are *Reverse Engineering* and *Forward Engineering*. Chikofsky et. al. [CC92] define these terms as follows,

> *Reverse Engineering* is the process of analysing a subject system to (i) identify the system's components and their relationships and (ii) create representations of the system in another form or at a higher level of abstraction.

[1] When we use the term *systems* we refer to industrial projects and development tools implemented in any language.

> *Forward Engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

This thesis is within the context of the reverse engineering of object-oriented systems. Our goal is to generate *high-level views* on object-oriented systems at different abstraction levels. We state that,

> A developer making changes or extensions to an object-oriented system must therefore understand the relationships among the classes or risk that seemingly innocuous changes break the implicit dependencies they play a part in.

In short, any unstructured change leads to *fragile systems* [MS98] that are difficult to extend or modify correctly.

2.2 Problems in Object-Oriented Systems

The goal of our approach is to identify and understand different dependencies among classes with the static analysis of object-oriented code. This is not a trivial task because in most of the cases either documentation is outdated or insufficient or the code presents implicit information or the code is implemented using bad styles of code programming. Before diving into these different problems that we find in the source code, we need to define the main terms used in our approach. It is important to remark that these definitions are constrained to the context of our work.

2.2.1 Terminology

The main terms we define are *dependency* and *explicit* and *implicit* dependency.

Dependency. An object A depends upon another object B, if it is possible that a change to B implies that A is affected or also needs to be changed, *i.e.*, dependency between a client and a server.

Explicit Dependency. A *dependency* between two or more objects is *explicit* when it is precisely and clearly expressed without ambiguity in the source code, *i.e.*, definition of a direct subclass (in Smalltalk we use the keyword `superclass` or in Java we use the keyword `extends`).

Implicit Dependency. A *dependency* between two or more objects is *implicit* when it can be implied from the source code though is not directly expressed, *i.e.*, chain of superclasses of a new defined class.

2.2.2 Common Problems in Object-Oriented Code

When reverse-engineering an object-oriented system, the first step a developer must perform is to get a mental model of the source code [SFM99]. With this first contact, he should be able to understand which are the different objects and different collaborations and relationships that determine the dependencies between them. The dependencies play a part in implicit contracts imposed in the system [SLMD96]. Thus any change in the source code should not break any of these contracts or add new unexpected ones.

Unfortunately building this mental model is not a trivial task because in most of the cases not all the dependencies between the objects are explicit in the system. These meaningful dependencies are determined by different building mechanisms, such as *reuse, delegation, data encapsulation, dynamic binding, inheritance* and *polymorphism*. Let's see briefly how these building mechanisms work and —when possible— identify some examples when they represent an *explicit* or *implicit* dependency between objects.

- **Class inheritance** is the mechanism to define a new class in terms of one or more parent classes. It means that the behavior and data associated with child classes are always an extension of the properties associated with parent classes. A subclass has all the properties of the parent class, and others as well.

  Example: The definition of a class in terms of one or more superclasses is an *explicit* dependency meanwhile all the chain of superclasses and inherited behavior and state of a class is an *implicit* dependency.

- **Delegation** is the mechanism that lets an object delegate to another object whatever behavior the first can not handle.

  Example: The delegation of a behavior in a method is an *explicit* dependency meanwhile all the chain of delegates is a *implicit* dependency.

- **Dynamic binding** is the mechanism to select lately the method until execution time. It has two main aspects: determine the object (and the type), and look up in the chain of superclass for the method.

  Example: The method lookup made by a chain of superclasses is an *implicit* dependency.

- **Data encapsulation,** sometimes referred to as data hiding, is the mechanism whereby the implementation details of a class are kept hidden from the user. The user can only perform a restricted set of operations on the hidden members of the class by executing special functions commonly called methods.

- **Polymorphism** is used to describe a variable that may refer to objects whose class is not known at compile time and which respond at run time according to the actual class of the object to which they refer. The polymorphism is shown as: (1) a variable holding a value drawn from a group of types, (2) a name associated with several different method bodies, and (3) a single method with polymorphic variables [2] as parameters

As we see each of these building mechanisms defines different kinds of dependencies. Even when these mechanisms are used correctly, several problems can appear [WH92]:

- Although dynamic binding is one of the most flexible mechanisms in object-oriented systems, the tracing of dependencies is difficult to grasp.
- The code for any given task is usually dispersed in several methods [NR89]. When using delegation, understanding a single line of code requires tracing a chain of method invocations through several different object classes and up and down the object hierarchy.
- When using inheritance, it has been observed that it may be difficult to find, for example, the right class to use for a group of objects out of Smalltalk's different classes of Collection [NR89]. Thus the developer may have some problems in finding where different functions are carried out, either to reuse them or to modify them.

Apart from these problems, other ones are identified and are due to misuse or overuse of the different mechanisms [Bud91]. In these cases, either unnecessary or complex dependencies are created or the dependencies do not exist at all.

- *Classes that make direct modifications to other classes:* Behavior that leads to the modification of data contained in another class are a violation of the encapsulation. This violation leads to unnecessary hidden dependencies between classes.
- *Classes with too much responsibility:* Classes with too much responsibility need to learn to delegate some of their responsibility to subordinate or helper Classes. Often a portion of the class behavior can be abstracted out and assigned to a helper class [BMMM98].
- *Classes with no responsibility:* A class with no responsibility serves no function, and usually can be eliminated in a manner that improves the design [FBB+99].

[2] A *polymorphic variable* is a variable declared as one class that can hold values from subclasses

- *Classes with unused responsibility:* A class with responsibilities that are not used serves no function either.
- *Misleading names:* This problem can be provoked by the wrong use of polymorphism. If a system contains several implementation of the same message with significantly different effects, a developer can be misled in interpreting the code, and eventually introduce errors when changes are made.
- *Unconnected responsibilities:* This problem occurs when a class has a collection of responsibilities that are not connected by data, functionality, or any other obvious binding.
- *Inappropriate use of inheritance:* This problem occurs when the relationships between class and superclass is not "is-a", or when the class can not inherit useful behavior from the superclass.
- *Repeated functionality:* This problem occurs when code is duplicated in two or more classes, instead of being abstracted into a common superclass.

From our experience in analyzing the source code [Aré03, ADN03, ABN04], we have identified five main features of these problems:

1. The presence of dependencies is obscure when we have overuse or misuse of the building mechanisms in object-oriented systems.
2. The problems occur several times in a system, *i.e.*, repeated functionality can appear in different parts of software in a system.
3. The problems do not occur isolated in a system, *i.e.*, a class can have too much responsibilities and does not inherit useful behavior from the superclass.
4. The problems can appear at different levels in the code, *i.e.*, repeated functionality is a problem at the application level, inappropriate use of inheritance is a problem at the class hierarchy level and unconnected responsibilities is a problem at the class level.
5. No tool is able to detect the recurring occurrences of these dependencies.

These features define which are the different constraints that we must solve to be able to build a mental model of the source code.

2.3 Goals of our Approach

Based on the characteristics of the problems in source code, the goals of our approach are the following ones:

- Identify dependencies that are implicit and make them explicit to analyze the source code.
- Show that the dependencies are not isolated in the system, and there are dependencies occurring with other ones.
- Show that the groups of dependencies occur several times in a system, and they can be identified as *patterns* in the system.
- Show that the dependencies occur at different abstraction levels. In the specific context of our work, we analyze applications at class-level, class hierarchy-level and complete application level.
- Generate *high-level views* at different abstraction levels that help the software engineers cope with the complexity of software development.
- Show that the *high-level views* are interconnected views between the abstraction levels
- Develop three analysis tools based on a conceptual clustering technique called *Formal Concept Analysis* to generate and analyze the high-level views.

The evaluation of this approach is driven by the following questions:

General Goals - Considering the use of FCA

- Is FCA an easy-to-use clustering technique in software reengineering?
- Is FCA scalable considering the amount of information we could have in big systems?
- What is the complexity time of the FCA?
- Is any limit in the use of the technique?
- Is the interpretation of the results an automatic process ?
- Does FCA identify known and unknown dependencies?

Specific Goals depending on Abstraction level of Analysis

- Class-Based Approach
  - How does the technique help in understanding the inner workings of a class?
  - Is there a limited number of X-Ray Views in a class?
  - Does FCA discover new dependencies in the class?
- Hierarchy-Based Approach
  - How does the technique help in discovering schemas introduced in a class hierarchy?
  - Do the schemas show new dependencies in the class hierarchy?
  - Can FCA help to identify situations where the developer could apply reverse engineering tasks?
- Application-Based Approach
  - How does the technique help in discovering patterns introduced in a complete system?
  - Does the technique discover know and unknown patterns?
  - How do the patterns help to understand a system ?

2.4 Reverse Engineering: State of the Art

Some reverse engineering approaches cope with the problem of building *high-level views* to generate the initial mental model of the system. Following we describe some of them, and we identify some limitations appearing in each case.

- **Reading existing documentation and source code** [Dek03][HCIM02][DDN02]. This task is feasible when the systems are small or for small pieces of software. In large applications, this task is impossible because only code reading can take weeks considering we have a well-designed system. Regarding existing documentation, in most of the cases it is out-of-date or obsolete.
- **Analyzing Execution Traces** [Ric02], [JR97]. The use of dynamic analysis in a system can reveal which is the behavior of a system when it is running. However, it has some important drawbacks in terms of scalability and interpretation. Regarding the scalability, it is impossible to generate all the execution traces produced by a system. Regarding the interpretation, if the developer does not choose the right execution traces to analyze, he can lose important information about the system.
- **Interviewing users and developers.** This task can be important to reveal domain knowledge implemented and not explicit in the system. But the information can be subjective and difficult to formalize. Additionally, a system is implemented during several years and in a group of developers. It is difficult to find a developer that participated in all the project development. And thus, some knowledge is lost when the developers leave the organizations.

- **Tool support.** The tool support is basically the most important issue in reverse engineering. Any tool that can abstract the developer from the source code helps. Depending on the information he wants to obtain, the support used can be visual one such as Rigi [Mül86], ShrimpViews [SM95] and CodeCrawler [Lan03], or just code analyzers such as query engines or slicers.
- **Analysis of Version History** [GDL04] [JGR99]. This is still a young research field, where the developers analyze the changes made in the software to predict future changes and avoid bad-design practices in the future versions.
- **Use of metrics** [FP96]. In most of the cases, metrics are used to assess the quality of source code by computing various metrics to detect specific characteristics, such as cohesive classes, coupling with other parts of the system.

We see that all the approaches are useful in identifying the initial fingerprints (mental model) of a system. But they introduce some limitations. Some of them are *ad-hoc* approaches, such as interviewing users and developers or reading documentation and source code. In both cases, some knowledge about the system is lost. In all the approaches, the maintainers work with already known dependencies in the system. They do not identify new or implicit dependencies, or grouping of them. The clustering techniques are an alternative to cope with the grouping of characteristics of a system and building *high-level views* . We describe them in detail in the next section.

2.5 Software Clustering

Clustering techniques have been used in many disciplines to support grouping of similar objects of a system. The definition given in Sharma [Sha96] is: *Clustering analysis is a technique used for combining observations into groups or clusters such that:*

- *Each group or cluster is homogeneous or compact with respect to certain characteristics. That is, observations in each group are similar to each other*
- *Each group should be different from other groups with respect to the same characteristics; that is observations of one group should be different from the observations of other groups*

Therefore the primary objective is to take a set of objects and characteristics with no apparent structure and impose a structure upon them with respect to a characteristic.

2.5.1 Common Problems for Clustering Techniques

To apply any clustering techniques, we need to address the following problems:

- **Data Representation Extraction:** This is a process to extract the most important properties from the data that we are analyzing. This process may need to transform some existing data to a new calculated data and feed it as input to a particular clustering technique.
- **Calculate the data similarities:** This is a process to calculate which attributes are fulfilled or not by the data.
- **Grouping:** Depending on a chosen technique, data will be grouped to create clusters.

2.5.2 Requirements for Clustering Techniques

A good clustering technique need to satisfy [HK00]:

- **Scalability:** The algorithm should be able to deal with big sets of data. A good clustering technique will allow new data be inserted and it will dynamically allocate that new data into an appropriate cluster.
- **Minimum input from user:** Partitioning algorithms require different input from the user (*i.e.*, number of clusters). A good clustering technique may need to eliminate some inputs from users
- **Handling noise data:** A good clustering technique can eliminate most noise data or outliners during clustering processing.
- **Acceptable computation time:** If a technique takes too much time to finish on a large data set, it may make no sense to apply the clustering result anymore.
- **Sensitivity to the order of data objects:** Some techniques require the data must be in order.
- **Interpretability:** Users must be able to understand the clustering result and be able to use it. That is, the clusters should have semantical meaning.

2.5.3 State of the Art

Clustering techniques can be applied to software during various life-cycles phases. Most of the approaches attempt to provide solutions in restructuring legacy systems. The existing literature can be divided in (1) applications of a specific clustering algorithm in a system, and (2) comparison of different clustering approaches and evaluate them based on specific characteristics. In spite that FCA is a clustering technique, in this state of the art we do not consider it in our analysis. We have made a detailed state of the art about applications of FCA in software engineering in Chapter 3, because this technique is one of the *cornerstones* of our approach.

Within the first category *"Application of a specific clustering algorithm"* we have found that:

- Belady et al. [BE81] present an approach that automatically clusters a software system in order to reduce its complexity. In addition, they provided a measure for the complexity of a a system after it has been clustered. All the information is extracted from the system's documentation.
- Hutchens et al. [HB85] perform clustering based on *data bindings*. A data binding was defined as an interaction between two procedures based on the location of variables that are within the static scope of both procedures. Based on the data bindings, a hierarchy is constructed from which a partition could be derived. Another additional features are (1) they compared their structures with the developer's mental model with satisfactory results; and (2) they evaluated the *stability* of the system, focusing on what happened with the clustering when changes are done.
- Schwanke et al. [SAP89][SP89][Sch91] work on the "classic" low-coupling and high-cohesion heuristics by introducing the "shared neighbors" technique, in order to capture patterns that appear commonly in software systems. His "maverick analysis" enabled him to refine a partition by identifying components that happened to belong to the wrong subsystem, and placing them in the correct one. However, his approach was never tested against a large software system.
- Choi et al. [CS90] present an approach to finding subsystem hierarchies based on resources exchanges between modules. The approach ability to scale up was questionable because of complexity of their algorithm.
- Müller [MU90] [Mül93] introduces a semi-automatic approach to help a designer perform clustering on a software system. He introduces the important principles of *small interfaces* (the number of elements of a subsystem that interface with other subsystems should be small compared to the total number of elements in the subsystem) and of *few interfaces* (a given subsystem should interface only with a small number of the other subsystems).
- Neighbors [Nei96] attempts to identify subsystems with the ultimate goal of manual extraction of reusable components. He looked at compile-time and link-time interconnections between components and tried different approaches. The approaches were based on naming and on reference context.

- Anquetil et al. [AL97] look at the names of the resources of the system to produce a clustered system. But this approach has a drawback relying on the developers' consistency with the naming of their resources.
- Mancoridis et al. [MM98] treat clustering as an opitmization problem and use genetic algorithms to overcome the local optima problem of "hill-climbing" algorithms, which are commonly used in clustering problems. They implemented a tool called Bunch [MMCG99] that can generate better results faster when users are able to integrate their knowledge into the clustering problems. They also show how the subsystem structure of a system can be maintained incrementally after the original structure has been produced.
- Lung [Lun98] shows two examples of how the clustering technique is used to support software architecture restructuring to minimize coupling. The first example is an empirical study of a legacy system where the restructuring is based on use cases. The second example is an initiative idea of identifying possible addition of design patterns in the system.
- Xu et al. [XLZS04] present an approach to program restructuring at the functional level based on the clustering technique with cohesion as the main concern. The approach focuses on automated support for identifying ill-structured or low cohesive functions and providing heuristic advice in both development and evolution phases. The empirical observations show that the heuristic advice provided by the approach can help software designers make better decision of why and how to restructure a program.

Within the second category *"Comparison of existing clustering techniques"* we have found that:

- Wiggerts [Wig97] presents a general overview of how clustering algorithms are a good starting point for the remodularization of software. In his work, he evaluates two main issues that imposes a structure which will satisfy the constraints of a good modularization: the choice of an algorithm and the criteria of classifying for good clusterings (known as *similarity*). From the categorization of the algorithms in: *graph theoretical algorithms* ([Gor81], [vR79], [Rog71], [Ros69], [vL93], [BS91]), *construction algorithms* ([vL93], [Wis69], [GL70]), *optimization algorithms* ([And73], [Eve74], [AB84], [Mac67], [KR90], [BH65]) and *hierarchical algorithms* ([KR90], [Ste92]), he chooses the most suitable characteristics from the different categories to build algorithms which are suited for the remodularization of legacy systems and the classification of their components. Within the classification criteria, he introduces the idea of *features* that characterize an object and evaluates different *similarity* measures which compute the similarity between objects based on the scores on selected features: *distance measures* [Eve74], *association coefficients* [SS73], [And73], [KR90], *correlation coefficients* [AB84], [KR90] and *probabilistic similarity measures* [AB84],[SS73].
- Tzerpos et al. [TH98] propose a survey of approaches to the clustering problem from researchers in the software engineering community. They present clustering techniques in other disciplines, and argue that their usage in a software context [JD88] could lead to better solutions to the software clustering problems.
- Anquetil et al. [AL99] present a comparative study of different hierarchical clustering algorithms and analyze their properties with regard to software remodularization: how the entities are *described*, how *coupling* between the entities is computed and what *algorithm* is used. From their experiments in file clustering, they found that these kinds of algorithms can be used to get different partitioning of the system at different level of abstraction.
- Koschke [Kos00] presents a framework to evaluate different clustering techniques for component recovery in systems to analyze their strengths and weaknesses in comparison with other techniques. The main goal is to establish an accepted benchmark suite and standard evaluation method of comparing different techniques.
- Mitchell et al. [MM01] propose a comparison of different clustering approaches applied on a same system. He measures the differences between the results using two similarity measurements. They also provide some suggestions on how to identify and deal with source code components that tend to contribute to poor similarity results.

Most of these techniques search for identifying *highly cohesive* and *loosely coupled* groups based on pieces of software. Although the grouping of objects is a feature fulfilled by these approaches, they only show how grouping can be done, but any semantical meaning for the groups is inferred with the results of the clusters. With Formal Concept Analysis, we cope with this drawback, and we show how to perform it in each application of our approach in the following chapters.

2.6 Conclusions

In this chapter, we have cited which are the different problems that source code presents when several object-oriented building mechanisms are used. We have identified the main features of these problems. In most of the cases,

1. The dependencies are implicit,
2. The dependencies do not appear isolated in the system,
3. The dependencies appear several times in the system,
4. The dependencies appear at different abstraction levels of a system, and
5. No tool exists to detect these dependencies.

We have also show how reverse engineering and clustering approaches solve partially the problem of generating *high-level views* to have the first contact with a system. We have seen that,

1. In the case of reverse engineering approaches some of them are *ad-hoc* and all of them work with already known dependencies in the system. They do not identify new or implicit ones, or grouping them.
2. Software clustering is an alternative to get groups of dependencies, but in all the cases the approaches show how grouping can be done, but a semantical meaning for the groups is missing.

Chapter 3

Formal Concept Analysis in High-Level Views

This thesis is about development of an approach to analyze object-oriented source code based on identifying implicit dependencies and generate *high-level views* at different abstraction levels. This approach is supported by a methodology and a framework tool over which we have built three tools to analyze the source code at class-, class hierarchy- and application levels. In this chapter we introduce in detail the basics of our approach based on Formal Concept Analysis, and what are the different issues that the developer must take into account when using the approach to generate the *high-level views* . We also summarize which are the existing FCA approaches developed to cope different problems at lifecycle process of a system. The mathematical background of FCA is not included in this chapter. The reader interested in knowing the formal features of FCA should consult Appendix B.

3.1 Introduction

Formal Concept Analysis is a clustering technique for discovering conceptual structures in data. These structures allow the discovery and analysis of (complex) dependencies within the data. We have developed our approach and the tool support based on FCA.

3.2 Overview of the Approach

In this section we describe the methodology of a general approach to use FCA to build tools that identify recurring sets of dependencies in the context of object-oriented software reengineering. Our approach conforms to a pipeline architecture [BMR+96] in which the analysis is carried out by a sequence of processing steps. The output of each step provides the input to the next step. We have implemented the approach as an extension of the *Moose* reengineering environment [DGLD04].

This methodology is supported by ConAn framework, a tool implemented in VisualWorks [Vis03]. The reader interested in details about this tool should consult the Appendix A.

The processing steps are illustrated in Figure A.2. We can briefly summarize the goal of each step as follows:

- *Model Import:* A model of the software is constructed from the source code.
- *FCA Mapping:* A FCA Context (Elements, Properties, Incidence Table) is built, mapping from metamodel entities to FCA elements (referred as *objects* in FCA literature) and properties (referred

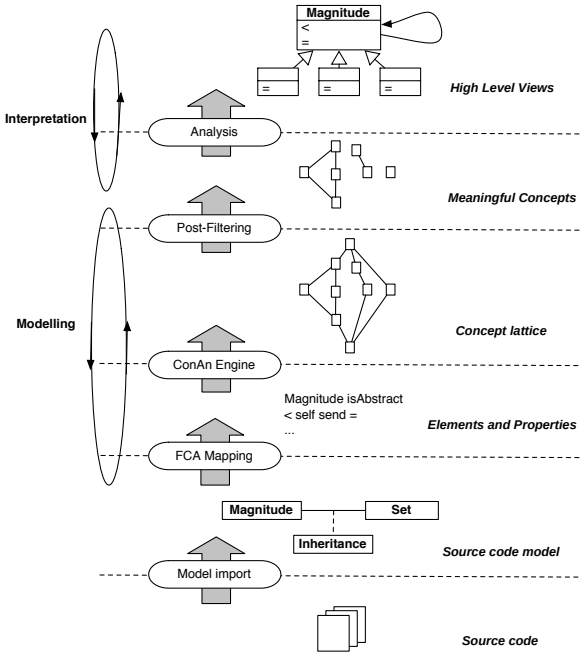

Figure 3.1: The overall approach

as *attributes* in FCA literature) [1].

- *ConAn Engine:* The concepts and the lattice are generated by the ConAn tool.
- *Post-Filtering:* Concepts that are not useful for the analysis are filtered out.
- *Analysis:* The concepts are used to build the high-level views

A key aspect of our approach is that one must iterate over the modeling and the interpretation phases (see Figure 3.1). The *modeling* phase entails a process of experimentation with smaller case studies to find a suitable mapping from the source code model to FCA elements and properties. A particular challenge is to find a mapping that is efficient in terms of identifying meaningful concepts while minimizing the quantity of data that must be processed.

The *interpretation* phase is the other iterative process in which the output of the modeling phase is analyzed in order to interpret the resulting concepts in the context of the application domain. The useful concepts can then be flagged so that future occurrences can be automatically detected. As more case studies are analyzed, the set of identifiably useful concepts typically increases up to a certain point, and then stabilizes.

From our experiences, there are two main participants in the approach: the *tool builder* and *the software engineer*. The *tool builder* builds the FCA-based tool to generate the high-level views, and the *software engineer* uses the results provided by the tool to analyze a piece of software. Both of them work together in the *modeling* and *interpretation* phase of the approach, because *software engineer* has the knowledge of analyzing a system and the *tool builder* can represent this knowledge in the tool.

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

3.3 FCA Applied in Software Engineering

FCA has been used in different phases of the process of software engineering. In Tilley et al. [TCBE03] the authors present a broad overview by describing and classifying different approaches in this field. The classification comprises two main categories: approaches used in *Early Phase Activities* and in *Software Maintenance*. In this section we complement the approaches mentioned in Tilley et al. [TCBE03] with new publications appeared in 2004. We give more details about approaches introduced in the second category because they are closer to the work developed in this thesis.

Early Phase Activities are all the activities that occur before the system is implemented

- *Requirement Analysis*
  FCA approaches in this category support the user by gathering and organize (semi) automatically the requirements. Düwel et al. [DH98][Düw99][DH00] and Böttger et. al. [BSR+01] develop an FCA-approach to reconcile differences in viewpoints of same use cases described in different ways by stakeholders requirements. Lattices allow to compute the closeness between viewpoints and to test when they are moving towards a shared viewpoint. Similarly, Richards et. al. [RB02a] [RB02b][RBF02a][RB02c][RBF02b] develop an approach to identify objects and classes based on use cases. They consider that FCA is an useful tool to structure and formalise conceptual thinking. In all the cases they consider that the approach is semi-automatic and help the communication between developers and customers of the system.
- *Component Retrieval Software*
  Lindig describes a retrieval system that could be used for retrieving software components from a library indexed by keywords [Lin95].
- *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 [Fis98].
- *Visualizing Z Specification via FCA*
  Z is a state based formal method that exploits the theory and first order predicate logic [Spi89]. In Tilley et. al. [Til03] they use ToscanaJ to conceptually navigate and explore a Z specification using FCA and retrieve relevant parts.
- *Generalization Level in UML Models*
  Dao et. al. [DHHaV04] propose an *iterative cross generalization* (a FCA-based methodology) which processes several mutually related formal contexts and sketches its application to UML class diagram restructuring.

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

- *Dynamic analysis*
  Ball [Bal99] examines test coverage while Bojic et al. [BV00] and Eisenbarth et. al. [EKS01a] [EKS01b] [EKS03] recover software architectures related to use cases identifying features scattered in different parts of the code. In Tonella et. al. [TC04] they identify possible candidates aspects relating execution traces and computational units (procedures, class methods) in a system.
- *Application to legacy systems*
  Snelting et al. [Sne96][KS94][FLS95] used FCA to analyze the preprocessor commands in legacy C programs in order to examine the configuration structure. van Deursen et. al. [vDK99] and Kuipers et. al. [KM00] compare the use of FCA for grouping fields within a large legacy COBOL program to that of hierarchical clustering. In both approaches they combine the use of type inference with FCA. Canfora et. al. [CCLL99] follow a similar approach but are interested in organizing a legacy COBOL system into components suitable for distribution via CORBA. Linding et al. [LY97] develop also the idea of identifying candidates modules in Modula-2, Fortran and COBOL. The task deriving object-oriented models from legacy systems written in C has been considered by Sahraoui et. al.

[SMLD97], Siff et. al. [SR97] and Tonella et. al. [Ton01]. 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*
  In reengineering class hierarchies, Godin et al. [GHRV02] categorize the existing works according to the structure of the output hierarchy of the approaches. Missikoff et. al. [MS89], Rundensteiner [Run92], Yahia et. al. [AYLCB96] and Snelting et. al. [ST97][ST98][Sne98] use *Galois (concept) lattice* as a final result of their analysis. Godin et. al. [GM93] [GMM+98], Dicky et. al. [DDHL95] [DDHL96], Huchard et. al. [HDL00], Cook [Coo92], Moore [Moo96] and Chen et. al. [CL96] use *Galois sub-hierarchy* [2] as a final results of their hierarchy.
  The result of all these approaches is again a class hierarchy built from the concept lattice in the former cases, and directly inferred from Galois sub-hierarchy in the last cases. This new class hierarchy is guaranteed to be behaviorally equivalent to the original hierarchy, but in which each object only contains the members that are required. In all the approaches, the algorithm to build the output hierarchy can be proved correct as it is done for CERES in Leblanc [Leb00]. They aim at improving the factorization while preserving the specialization relationships. They converge toward a normalized model of a class hierarchy. The methods are primarily intended as a tool for finding imperfections in the design of class hierarchies, and can be used as the basis for tools that largely automate the process of reengineering such hierarchies.

- *Analysis of classes*
  Analyzing how the fields are used by the methods in a JAVA class, Dekel et al. [Dek03] uses a lattice to reason about the interface and structure of the class and find errors in the absence of source code. They claim that their technique serves as a heuristic for automatic feature categorization, enabling it to assist efforts of re-documentation.

- *Conceptual analysis of software structure*
  Tonella et. al. [TA99] attempt to recover the structure of 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.

- *Software Testing*
  Sampath et. al. [SMSP04] cluster user sessions and maintain and update a reduced test suite for user session based testing of web applications. With these clusters, they avoid collecting large user sessions data and provide some scability for the approach.

3.4 Our Approach in Depth

In Section 3.2 we have introduced briefly the different steps of the architecture of our approach. We will now describe each processing step in detail and outline the key issues that must be addressed in order to apply the approach.

[2] By analogy with normalization for database design, the concept lattice can be considered as a kind of normal form for the design of class hierarchies. The lattice shows all potentially useful generalizations. Some of the generalizations of the concept lattice are empty in that they do not possess their own attributes or objects: all their attributes appear in at least one super-concept (inheritance) and dually, all their objects appear in a sub-concept (extension inclusion). These concepts could be eliminated without loss of information thus leading to a structure called a Galois sub-hierarchy which corresponds to the union of what is called the sets of attribute concepts and object concepts of the concept lattice. This structure is not necessarily a lattice but, when interpreting its nodes as classes, it is maximally factored, consistent with specialization, while defining a minimal number of classes. It could also be considered as another kind of normal form. As for normalization, in practice, there might be reasons to introduce deviations from these ideal structures but the decision process should be made in a controlled manner by using the normal forms as a conceptual framework [GHRV02].

Model Import

Description: Our first step is to build a *model* of the application from the source code. For this purpose we use the *Moose* reengineering platform, a research vehicle for reverse and reengineering object-oriented software [DGLD04]. Software models in *Moose* conform to the FAMIX metamodel [TDDN00], a language-independent metamodel for reengineering. *Moose* functions both as a software repository for these models, and as a platform for implementing language-independent tools that query and manipulate the models.

Issues: In this step the most important issue is how to map the source code to metamodel entities. The main goal of this step is to have a language-independent representation of the software. In our specific case, we use the FAMIX metamodel, which includes critical information such as method invocations and attributes accesses. The tool builder can, however, choose any suitable metamodel.

FCA Mapping

Description: In the second step, we need to map the model entities to *elements* and *properties*, and we need to produce an *incidence table* that records which elements fullfil which property. The choice of elements and properties depends on the view we want to obtain.

Issues: This is a critical step because several issues must be considered. Each of these issues is part of the iterative *modeling* process.

- *Choice of Elements:* First we must decide which metamodel entities are mapped to FCA elements. This is normally straightforward. In most cases there are some particular metamodel entities that are directly adopted as FCA elements (*e.g.*, classes, methods in case of *XRay Views* (Chapter 4) and *Hierarchy Schemas* (Chapter 5)). Alternatively, a FCA element may correspond to a set of entities (*e.g.*, a set of collaborating classes in case of *Collaboration Patterns* (Chapter 6)).
- *Compact Representation of Data:* In some cases, a naïve mapping from metamodel entities to FCA elements may result in the generation of massive amounts of data. In such cases, it may well be that many elements are in fact redundant. For example, if method invocations are chosen as FCA elements, it may be that multiple invocations of the same method do not add any new information for the purpose of applying FCA. By taking care in how FCA elements are generated from the metamodel, we can not only reduce noise, but we can also reduce the cost of computing the concepts.
- *Choice of Properties:* Once the issue of modeling FCA elements is decided, the next step is to choose suitable properties. Well-chosen properties achieve the goal of distinguishing groups of similar elements. This means that they should neither be too general (so that most elements fulfill them) nor be too specific (so only few elements fulfill them).
- *Use of negative properties:* Nevertheless, in some cases the developer needs still more properties to distinguish the elements. But simply adding more properties may only increase the complexity of the approach. The use of "negative" information (built by negating existing properties) may help.
- *Single or Multiple FCA Contexts:* In some cases, multiple FCA contexts may be required to analyze the same set of software artifacts from different viewpoints.
- *Computation of properties or elements:* When building the FCA context of a system to analyze, there are two alternatives of FCA mapping. In an *one-to-one* mapping, the developer directly adopts metamodel entities and metamodel relationships as FCA elements and properties respectively. In a *many-to-one* mapping the developer builds more complex FCA elements and properties by computing them from the metamodel entities and relationships, meaning for example that a FCA element can be composed of several metamodel entities, or a FCA property must be calculated based on metamodel relationships between different entities. This issue is one of the bottlenecks in the total computation time of the approach, because the incidence table must be computed in this step and if the FCA property must be calculated, this time can also compromise the total computation time.

ConAn Engine

Description: Once the elements and properties are defined, we run the ConAn engine. The ConAn engine is a black-box component implemented in VisualWorks 7 which runs the FCA algorithms to build the concepts and the lattice. ConAn applies the Ganter algorithm [GW99] to build the concepts and our own algorithm to build the lattice. Both algorithms are specified in the Appendix B.

Issues: In this step, there are three main issues to consider

- *Performance of Ganter algorithm:* Given an FCA Context $C = (E, P, I)$, the Ganter algorithm has a time complexity of $O(|E|^2|P|)$. This is the second bottleneck of the approach because in our case studies the number of FCA elements is large due to the size of the applications. We consider that |P| is not a critical factor because in our case studies the maximum number of properties is 15.
- *Performance of Lattice Building Algorithm:* Our algorithm is the simplest algorithm to build the lattice but the time complexity is $O(n^3)$ where *n* is the number of concepts calculated by Ganter algorithm. This is the last bottleneck of the approach.
- *Unnecessary properties:* It may happen that certain properties are not held by any element. Such properties just increase noise, since they will percolate to the bottom concept of the lattice, where we have no elements and all properties of the context.

Post-Filtering

Description: Once the concepts and the lattice are built, each concept constitutes a potential *candidate* for analysis. But not all the concepts are relevant. Thus we have a *post-filtering* process, which is the last step performed by the tool. In this way we filter out meaningless concepts.

Issues: In this step, there are two main issues to consider:

- *Removal of Top and Bottom Concepts:* The first step in *post-filtering* is to remove the *top* and *bottom* concepts. Neither provides useful information for our analysis when each contains an empty set. (The intent is empty in the top concept and the extent is empty in the bottom concept).
- *Removal of meaningless concepts:* This step depends on the interpretation we give to the concepts. Usually concepts with only a single element or property are candidates for removal because the interesting characteristic of the approach is to find *groups* of elements sharing *common* characteristics. Concepts with only a single element occur typically in nodes next to the bottom of the lattice, whereas concepts with only one property are usually next to the top of the lattice.

Analysis

Description: In this step, the software engineer examines the candidate concepts resulting from the previous steps and uses them to explore the different *implicit* dependencies between the software entities and how they determine or affect the behavior of the system

Issues: In this step, there are several issues to consider. All of them are related to how the software engineer interprets the concepts to get meaningful or useful results.

- *Concept Interpretation based on Elements or Properties:* Once the lattice is calculated, we can interpret each concept $C = (\{E_1 \dots E_n\}, \{P_1 \dots P_m\})$ using either its elements or its properties. If we use the properties, we try to associate a meaning to the conjunction of the properties. On the other hand, if we focus on the elements, we essentially discard the properties and instead search for a domain specific association between the elements (for example, classes being related by inheritance).
- *Equivalent Concepts:* When we interpret the concepts based on their properties, we can find that the meaning of several concepts can be the same. This means that for our analysis, the same meaning can be associated to different sets of properties and different concepts.

- *Supporting End-users:* If the results must be read by end-users (rather than by software engineers) then it may be necessary to introduce an additional abstraction level over that of the concept lattice in order to present the results in a form more appropriate to the user domain.
- *Automated Concept Interpretation:* The interpretation of concepts can be transformed into an automatic process. Once the software engineer establishes a meaning for a given concept, this correspondence can be stored in a database. The next time the analysis is performed on another case study, the established interpretations can be retrieved and automatically applied to the concepts identified. Once this process is finished, the software engineer must still check those concepts whose meaning has not been identified automatically.
- *Using Partial Order in the Lattice:* The concepts in the lattice are related by a partial order. During analysis, the software engineer should evaluate if it is possible to interpret the partial order of the lattice in terms of software relationships. This means that the software engineer should only not interpret the concepts but also the relationships between them.
- *Limit of using FCA as a grouping technique:* When additional case studies fail to reveal new meaningful concepts, then the application of FCA has reached its limit. At this point, the set of recognized concepts and their interpretations can be encoded in a fixed way, for example, as logical predicates over the model entities, thus fully automating the recognition process and bypassing the use of FCA.

3.5 Conclusions

3.5.1 Research Questions

Based on the approach, we are able to answer some of the research questions proposed in Chapter 2:

- *Is FCA an easy-to-use clustering technique in software reengineering ?*

  As we have seen in this chapter, FCA is an useful technique but it is not trivial to use. In each step, there are *critical issues* that determine how meaningful results will be in the final step: *Analysis*. The issues about building the FCA context(s) in *FCA Mapping* affect the performances of the algorithms (in the step ConAn *Engine*), filters and interpretation of the results (issues in the steps *Post-Filtering* and *Analysis*). According to our experience, when the choice of elements and properties is not the adequate one, the results are meaningless. It is the main reason that we have in our approach two iterative processes: *Modelling* and *Interpretation*. These two iterative processes help in finding a suitable mapping to reduce *noise* in the results and obtain *meaningful* concepts that are used in building the *high-level views* of a system.

- *Is FCA scalable considering the amount of information we could have in big systems ?*

  Based on our 3 abstraction levels of analysis (introduced in Chapters 4, 5 and 6) we believe that FCA is scalable and can be used with big amount of information (when we have acceptable computation times). We confirm this answer in the different analysis shown in the following chapters.

- *What is the time complexity of the FCA ?*

  In the step ConAn *Engine* we have indicated the complexity time of Ganter and *lattice-building* algorithms. We see that they represent two bottlenecks of our approach. If we do not restrict our analysis about complexity to the algorithms, we also consider that the total complexity time can be compromised when the FCA elements and properties must be calculated in terms of information provided by software artifacts (issue named as *Computation of Properties and Elements* in the step *FCA Mapping*).

- *Is there any limit in the use of the technique ?*

  So far, we do not believe that we have reached the limit in using the technique. We consider it as an issue in the step *Analysis*. When additional case studies in each approach fail to reveal new meaningful concepts, then the application of FCA has reached its limit.

- *Is the interpretation of the results an automatic process ?*

  The interpretation of the results is not automatic. We consider it as an issue in the step *Analysis*. The transformation can be transformed into an automatic process. Once the software engineer establishes a meaning for a given concept, this correspondence can be stored in a database. This process is followed in each application of case studies with all concepts that are not identified automatically by the database. The database increments its number of correspondences until no new meaningful concepts are identified. This is also an indication of the limit in using the approach.

- *Does FCA identify known and unknown dependencies ?*

  The answer to this question is detailed in the Chapters 4, 5 and 6.

3.5.2 Summary

In this chapter we have explained the details of our methodology and which are the different key issues the developer must take into when applying the approach. The methodology (supported by a tool named ConAn) can be considered as a general one because it can reproduced in any other tool that has FCA as a base tool. This methodology is the result of our experiments applied in three different abstraction levels. But it is applicable in other abstraction levels.

We have also summarized how existing FCA-based approaches are applied in different lifecycle of a system. The approaches show that FCA is a promising technique used in different lifecycles of a system. In all the cases, the main drawback is a lack of a methodology about how to build the context of the case studies to apply FCA. As we have seen in this chapter, it is not a trivial task.

Chapter 4

XRay Views: Supporting Class Understanding

Supporting classes understanding is a key task in software reverse engineering as classes are the cornerstone of the object-oriented paradigm and the primary abstraction from which applications are built. In this chapter we focus on the first abstraction level of a system: a class. We develop a methodology focused on understanding a class as an isolated development unit. Our analysis is based on the internal structure of a class and focus on how the methods call each other and access attributes. Instead of requiring the engineer to read code line-by-line, we provide three logically connected "XRay views" of classes: STATE USAGE, EXTERNAL/INTERNAL CALLS and BEHAVIOURAL SKELETON. Each of these XRay views is composed out of elementary dependencies between a set of methods and a set of attributes. Thus, these relationships show us the *internal contracts* of the state and the behavior of a class. In this way we support *opportunistic* understanding [LPLS96] in which the engineer understands a class iteratively by exploring the views and reading code.

4.1 Problems in Understanding Classes

Classes are the cornerstone of the object-oriented paradigm. They act as factories of objects and define the behavior of their instances. However, they are harder to understand due to several reasons [Dek03, LD01]:

1. Contrary to procedural languages, the method definition order in a file is not important [Dek03]. There is no simple and apparent top-down call decomposition, even if some languages propose the visibility notion (private, protected, and public). This drawback makes code understanding a difficult task because there is no predefined order to follow the method call graph.

2. The presence of late-binding leads to "yoyo effects" when walking through a hierarchy and trying to follow the call-flow [WH92].

3. Dynamic binding and polymorphism increase the number of potential dependencies within a program. Thus, the associations created through the use of polymorphism and dynamic binding usually mean that more than one class needs to be looked at (especially in the case of a class which is part of a deep inheritance hierarchy) to fully understand how the code works [DRW00]. Soloway et al. [SLL+88] has termed this phenomena as *delocalization* of the code.

4. Classes define state and the methods that act on this state. It is important to understand how the state is accessed, presented, if at all, to the class's clients, and how subclasses access this state.

4.2 Goals of XRay Views

Different features are important to understand the inner workings of a class. Some of this kind of information that an engineer would typically like to know about a class is:

- Which methods access any attribute, directly or indirectly
- Which groups of methods access directly or indirectly all the attributes or some subset of the attributes,
- Which methods are only called internally,
- Which methods/attributes are heavily used and accessed,
- How the methods and attributes collaborate.

Unfortunately all this information is implicit in the source code, and therefore cannot easily be teased out by a straightforward reading of the source. Our goal is to analyze groups of methods and attributes that collaborate together. But if we consider the number of attributes and methods we can find in a class, *mining* these elements –without clear criteria— can lead to an exponential number of groups that only makes the analysis more difficult. For this reason we generate a dedicated graph representation of the source code with the attributes' accesses and methods' calls of a class. With this graph, we analyze the class using FCA to detect different dependencies between sets of methods and attributes and we run our tool ConAn. Thus, *concepts* generated by FCA algorithm, are composed to build the *XRay Views*. We limit our approach to understanding a single class, without taking into account relationships to subclasses, superclasses, or peer classes. The reason to impose this limitation is that we want to explore the idea in small scenarios (before using it in complete applications) and analyze the impact of the methodology in the analysis. To help in the visualization and analysis of the results, we use *Class Blueprints* [LD01] as a complement to ConAn. A *Class Blueprint* is a semantically augmented visualization of the internal structure of a class, which displays an enriched call-graph with a semantics-based layout *i.e.*, methods are categorized based on simple heuristics and form layers grouping methods in the context of a call-graph starting from the public interface of a class.

Structure of the Chapter. This chapter is structured as follows: Section 4.3 provides a brief example of how FCA is applied in a class defining elements and properties based on attributes and methods of a class. We also show how we interpret the concepts of the lattice. Section 4.4 introduces the different case studies used to validate the XRay views and we present the 3 XRay views: STATE USAGE, EXTERNAL/INTERNAL CALLS and BEHAVIOURAL SKELETON in a *pattern-like* format. Section 4.5 analyzes several issues (introduced in Chapter 3) related to the application of the approach in class understanding. Section 4.6 provides some discussion points based on the experiments of generating *XRay Views*. Section 4.7 presents some related work to class understanding using other techniques. And final Section 4.8 we conclude the chapter and outline some future work.

4.3 Formal Concept Analysis in Class Understanding

At the class level, we apply FCA to identify concepts that correspond to the dependencies between the state and the behavior within a single class. We therefore build 4 CA contexts where we have:

- 2 FCA contexts with *methods* as elements and *attributes* as properties and the binary relation is the direct and indirect *accesses* relationships respectively.
- 2 FCA contexts with *methods* as elements and properties and the binary relation is the direct and indirect *invocations* relationships respectively

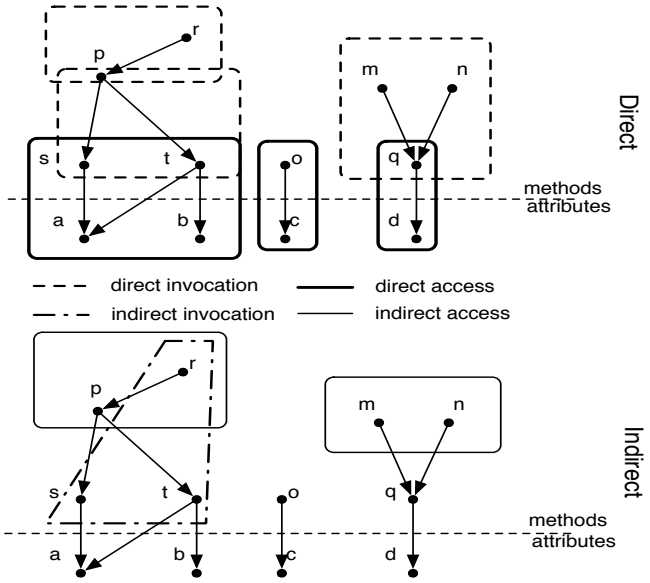

Figure 4.1: Attribute accesses and method invocations and the identified groups.

Let us see then which are the different properties between attributes and methods in a class we extract from the source code. These properties help us to identify the groups that we use to build the different XRay views.

4.3.1 Elements and Properties of Classes

Suppose a class has a set of methods $\mathcal{M}$ and a set of attributes $\mathcal{A}$. The basic properties we use are extracted from the source code as follows:

- m *accesses* x means that the method $m \in \mathcal{M}$ either directly reads/updates the value of attributes $x \in \mathcal{A}$ or uses a "getter/setter" method to access/modify the value of x.
- m *calls* n means that the method m calls the method n explicitly via a *self-call* [1].

In Figure 4.1 we see a graphical representation of a class with methods $\mathcal{M} = \{m, n, o, p, q, r, s, t\}$ and attributes $\mathcal{A} = \{a, b, c, d\}$. Here we have, for example, m *calls* q, r *calls* p, o *accesses* c, and they are represented with $m \rightarrow q$ and $o \rightarrow c$ respectively.

These properties express direct dependencies between entities. We are also interested in *indirect* dependencies, for example, m *accesses* d *indirectly* (which we write "m *accesses** d"). Indirect dependencies are important in revealing dependencies between methods and attributes, and are helpful in assessing the impact of changes in the class. We therefore define as well the following derived properties:

- m *calls** n if m *calls* m' and either m' *calls* n or m' *calls** n (*i.e.,* *calls** $= \cup_{i \geq 2}$ *calls* ^{i})
- m *accesses** x if m *calls* m' or m *calls** m', and m' *accesses* x (*i.e.,* *accesses** $= \cup_{i \geq 1}$ *calls* $^i \cdot$ *accesses*)

In the example, we see that p *calls* s and s *accesses* a, and consequently p *accesses** a.

Finally, we are sometimes interested to know when elements do *not* exhibit a certain property, so we introduce the following notation to express the negation of a relation:

[1] We do not focus on pointer analysis

- $e \neg R\, p$ if it is not true that $e\ R\ p$.

For example, $m \neg$ *accesses* c.

4.3.2 Properties among Groups

Since we are interested in dependencies occurring between *sets* of methods and attributes, we extend our properties to sets in the obvious way. Suppose that F and G are arbitrary subsets of the set of elements E. We define:

- $F\ R\ G$ means that each entity in F is related with each one in G, *i.e.*, $\forall e \in F, e' \in G, e\ R\ e'$.
- $F\ \overline{R}\ G$ means that the entities in F are *related exclusively* with those in G, *i.e.*, $\forall e \in E, e' \in G, e\ R\ e',\implies e \in F$ and conversely, $\forall e \in E, e' \in F, e'\ R\ e \implies e \in G$.

4.3.3 Interpretation

We introduce now the elementary groups based on which XRay views are built. Note that in each case we are interested in *all* of the participants of a given group. For example, below we define Collaborating Attributes, but we are interested not only in the attributes themselves, but also in the set of methods that access them. This holds for each example group listed below. We must remark that each group is a mapping of the concepts resulting from the 4 FCA contexts described previously. We illustrate each case with the example shown in Figure 4.1.

Direct Accessors: Direct accessors, readers or writers $M \subseteq \mathcal{M}$ of an attribute a are defined by non-exclusive relationships:

- M *accesses* $\{a\}$

This group provides us with a simple classification of the methods according to which attributes they use. In our example, $\{s, t\}$ *accesses* $\{a\}$.

Exclusive Direct Accessors: A method m is an *exclusive direct accessor* of a when m is the *only* method to access a directly. We are interested in the sets of exclusive direct accessors of an attribute:

- $M\ \overline{\textit{accesses}}\ \{a\}$

In our example, we see that $\{q\}\ \overline{\textit{accesses}}\ \{d\}$.

Exclusive Indirect Accessors: We consider a method m as an *exclusive indirect accessor* when it is the only method that accesses an instance variable using a *direct accessor* method of a specific attribute or calling a method that calls a *direct accessor* of an instance variable. It is represented as an exclusive relationship:

- $M\ \overline{\textit{accesses}^*}\ \{a\}$

This group distinguishes those methods that define the behaviour of a class without using at all the state from those that use the state of the class. In our example, we have $\{m, n\}\ \overline{\textit{accesses}^*}\ \{d\}$.

Collaborating Attributes: This group expresses which attributes are used exclusively by a set of methods:

- M $\overline{accesses^*}$ A

This group identifies groups of attributes working together in the class, and also which are the methods that work together using part of the state of the class.

In the example, we have only one set of collaborating attributes $\{t\}$ $\overline{accesses}$ $\{a, b\}$.

Stateful Core Methods: This group is a special case of *collaborating attributes* and expresses which methods access *all* the state of a class:

- M $\overline{accesses}$ $\mathcal{A}$

This group is interesting because it provides a guideline if all the attributes are being used in the core of the class, and providing a functionality to the class through a set of methods. In the example, there are no methods accessing the entire state of the class.

Collaborating Methods: This group expresses which methods uses the behaviour defined in the class. It is represented by an *exclusive dependency*:

- M $\overline{calls}$ M'
- M $\overline{calls^*}$ M'

This group helps us to identify the direct and indirect collaborations between groups of methods inside the class. In the example, $\{r\}$ $\overline{calls}$ $\{p\}$, $\{p\}$ $\overline{calls}$ $\{s, t\}$, $\{r\}$ $\overline{calls^*}$ $\{s, t\}$ and $\{m, n\}$ $\overline{calls}$ $\{q\}$.

Interface Methods: This group expresses which methods are not used at all inside the class. They are *pure* Interface Methods [2]. It is represented with an *exclusive dependency* as:

- $\mathcal{M}$ $\overline{\neg\, calls}$ M

M is the complete set of interface methods since there is no method in $\mathcal{M}$ that calls them, and there exist no other such methods. $\mathcal{M}$ $\overline{\neg\, calls}$ $\{r, o, m, n\}$ identifies the interface methods of our example class in Figure 4.1.

Externally Used State: This dependency expresses which interface methods are *direct accessors*:

- $\mathcal{M}$ $\overline{\neg\, calls}$ M and M *accesses* $\{a\}$

This group helps us to determine which methods are used as interface to the class and access directly the state of the class. In the example, only o provides externally used state, since $\mathcal{M}$ $\overline{\neg\, calls}$ $\{o\}$ and $\{o\}$ *accesses* $\{c\}$.

[2] In Smalltalk, there is no concept of *interface* compared to C++ or Java. All the methods are public and can be interface. It is possible to group the methods that behave as *public interface* in a method protocol. But the class clients can still call the *non-interface* methods. In typed object-oriented languages such as C++ or Java, it is possible to define a "pure interface" artifact consisting only of abstract method declarations — enabling a variable, typed to the pure interface, to reference objects belonging to any conforming classes — not restricted to any one part of the class hierarchy. In C++ such a pure interface is a *specification class* composed only of pure virtual functions, the *class* construct still being used as if it were a normal class. Multiple inheritance in C++ allows a specification class to act as a pure interface for a class anywhere in the class hierarchy. In Java a special language construct *interface* is provided consisting only of abstract methods. Although a class in Java can inherit from only one other class, it can additionally implement multiple interfaces [RHM00].

Stateless Methods: This group expresses which methods complement the *collaborating* ones, *i.e.*, which methods provides a service without calling any other methods or accessing the state of the class:

- $M = M_1 \cap M_2$, where $M_1 \overline{\neg \mathit{calls}}\ \mathcal{M}$ and $M_2 \overline{\neg \mathit{accesses}}\ \mathcal{A}$

There are no stateless methods in the example, since every method either calls another method or accesses some state.

4.4 XRay Views

An XRay view is a *combination* of groups that exposes specific aspects of a class. Based on the groups specified above, we now define three complementary XRay views: STATE USAGE, EXTERNAL/INTERNAL CALLS, and BEHAVIOURAL SKELETON. These three views address different, but logically related aspects of the behaviour of a class. STATE USAGE focuses on the way in which the state of a class is accessed by the methods, and exposes, for example, how cohesive the class is. EXTERNAL/INTERNAL CALLS categorizes methods according to whether they are internally or externally used, while BEHAVIOURAL SKELETON focuses on the way methods invoke each other internally.

To illustrate our approach, we present three Smalltalk classes — OrderedCollection, Scanner and UIBuilder — from the VisualWorks Smalltalk distribution [Vis03]. We chose these particular three classes because they are different enough in terms of size (shown in Table 4.1 and functionality, they address a well-known domain that the reader is certainly familiar with, and they show characteristic results of XRay view application. Here follows a brief description of these classes:

OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. The elements are accessible by external keys that serve as indices. This class has attributes firstIndex and lastIndex that index the first and last elements in the collection. Moreover OrderedCollection has an anonymous array-like attribute. Its behaviour is defined by 56 methods from which 24 redefine methods inherited from the superclass.

UIBuilder implements the Builder design pattern [GHJV95]. It is a complex class that is used to build user interfaces (windows and their subcomponents) according to declarative specifications provided by its clients. A UIBuilder is created and used at interface opening time by the client's interface opening method. UIBuilders use a special library of user interface components tailored for automatic user interface generation such as radio buttons, action buttons, and check boxes. UIBuilders can build and install composites of these components to any desired level of nesting. This class has 18 attributes and its behaviour is defined in 122 methods.

Scanner represents a traditional language scanner for the Smalltalk language. It scans a stream of Smalltalk tokens with a single look ahead. This class has ten attributes which refer to the source, to the current character, current token, current token type, a type table, and comments. Its behaviour is defined with just 24 methods which are procedurally-coded.

Class	HNL	Attributes	Methods
OrderedCollection	3	3	56
UIBuilder	1	19	122
Scanner	1	10	24

Table 4.1: Data about the classes (HNL indicates the level of inheritance)

We describe the three XRay views according to a common pattern: first we provide a *description*, then the *groups* used to build the view, and finally a *rationale* indicating the key aspects that the view can reveal.

For each view, we ran our analysis tool ConAn on the three classes. Then, we examined the resulting views by looking at and combining the groups presented in the "Used and Shown Groups" section of the view definition. We validated our findings by reading the source code opportunistically, and using *Class Blueprints* [LD01] as a complementary tool to visualize and annotate information about the found groups. As we said previously, a *Class Blueprint* visualizes the internal structure of a class with an enriched call graph. In the right pane of the Figure 4.2, we see the *Class Blueprint* of OrderedCollection. The nodes represent methods except the two rightmost ones representing attributes. The edges represent method invocations except the edges pointing to the rightmost nodes that represent accesses to the attributes.

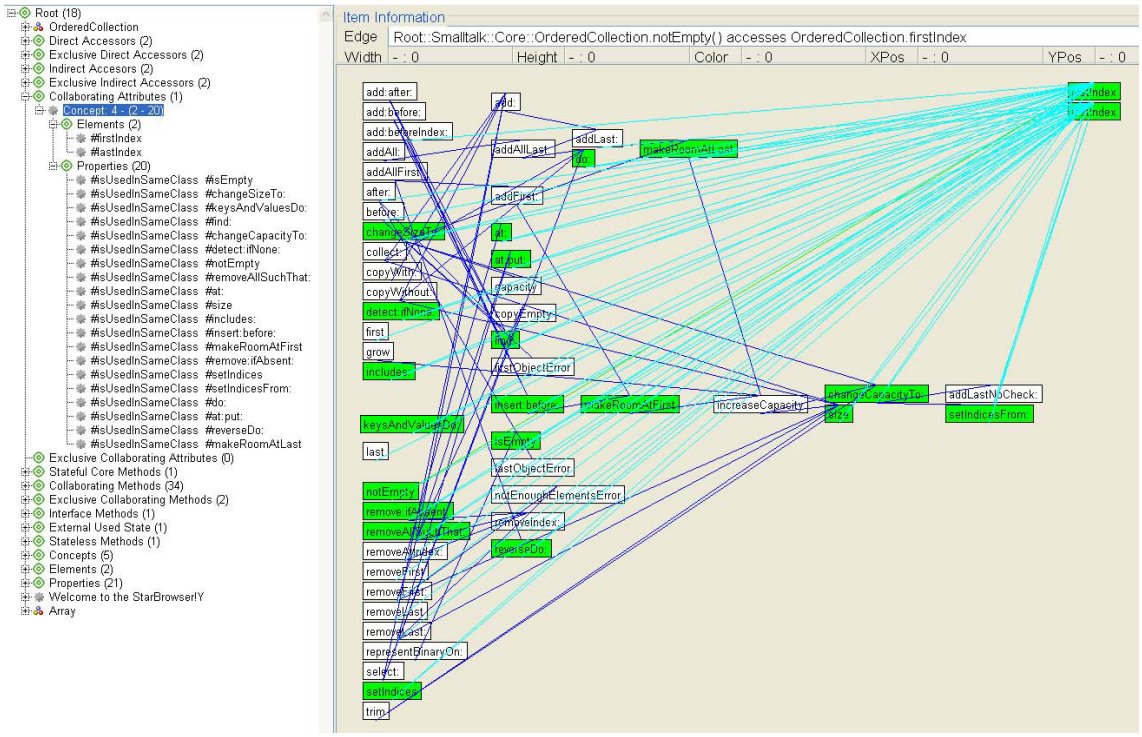

Figure 4.2: Group *Collaborating Attributes* of the XRay view STATE USAGE in OrderedCollection

4.4.1 XRay View: STATE USAGE

Description: Clusters attributes and methods according to the way methods access the attributes.

Used and Shown Groups: Exclusive Direct Accessors, Exclusive Indirect Accessors, Collaborating Attributes, and Stateful Core Methods.

Rationale: Objects bundle behaviour and state. To understand the design of a class, it is important to gain insight into how the behaviour accesses the state, and what dependencies exist between groups of methods and attributes. This view helps us to measure the cohesion of the class [BDW98] revealing if there are methods using the state partially or totally and if there are attributes working together providing different functionalities of the class.

Validation with OrderedCollection: Some of the groups used in STATE USAGE are the following ones:.

- {before, removeAtIndex:, add:beforeIndex:, first, removeFirst, removeFirst:, addFirst} $\overline{accesses}$ {firstIndex} represents the Exclusive Direct Accessors of firstIndex.
- {addLast:, copyWithout:, select:, trim, add:, representBinaryOn:, add:before:, increaseCapacity, collect:, grow, after:, add:after:, addAllLast:, addAll:, addAllFirst:, removeLast: } $\overline{accesses^*}$ {firstIndex } represents the Exclusive Indirect Accessors of firstIndex
- {makeRoomAtFirst, changeSizeTo:, removeAllSuchThat:, makeRoomAtLast, do:, notEmpty:, keysAndValuesDo:, detect:ifNone:, changeCapacityTo:, isEmpty, size, remove:ifAbsent:, includes:, reverseDo:, find:, setIndices, insert: before:, at:, at:put:, includes: } $\overline{accesses}$ {firstIndex, lastIndex } represents the Collaborating Attributes
- Stateful Core Methods = the same set as Collaborating Attributes

Similarly to the first two groups, we have groups of Exclusive Direct Accessors and Exclusive Indirect Accessors of lastIndex. In the Figure 4.2 we see the group of *Collaborating Attributes* as a concept and using *Class Blueprints* applied in OrderedCollection. In grey, we highlight the elements of the group *Collaborating Attributes* listed in the left part of the figure. The nodes in white are those accessing either firstIndex or lastIndex, but not both of them. As we can see each group determines a *subgraph* of the *Class Blueprint* of the class.

Before analysing the groups identified by this view, we posed the hypothesis that the two attributes maintain an invariant representing a memory zone in the third anonymous attribute. From the analysis we obtain the following points:

- We note that the attributes firstIndex and lastIndex have no getters or setters, so the state of the class is not exposed to clients.
- By browsing Exclusive Direct Accessors methods, we confirm that the naming conventions used help the maintainer to understand how the methods work with the instance variables, because we see that the method removeFirst accesses firstIndex and removeLast: accesses lastIndex respectively.
- The numbers of methods that exclusively access each attribute are very similar, however, we discover (by inspecting the code) that firstIndex is mostly accessed by readers, whereas lastIndex, is mostly accessed by writers.
- It is worth noting that Collaborating Attributes are accessed by the same methods that are identified as Stateful Core Methods. This situation is not common even for classes with a small number of attributes, and reveals a cohesive collaboration between the attributes when the class is well-designed and gives a specific functionality, in this specific case, dealing with collections.
- We identified 20 over 56 methods in total that access systematically *all* the state of the class. By further inspection, we learned that most of the accessors are readers. There are only five methods, makeRoomAtFirst, makeRoomAtLast, setIndices, insert:before:, and setIndicesFrom:, that read and write the state at the same time. More than half of the methods (33 over 56) directly and indirectly access both attributes. This confirms the hypothesis that the class maintains a strong correlation between the two attributes and the anonymous attribute of the class.

Validation with UIBuilder: The results are quite different compared to those obtained for OrderedCollection.

- We find getters and setters for each attribute.
- If we consider only the methods that access directly the attributes, we can classify the attributes into two groups: (a) attributes that are accessed by their getter and setter, and one or two additional methods (converterClass, windowSpec, spec, decorator, cacheWhileEditing); and (b) attributes that are accessed by several methods. Note that the view EXTERNAL/INTERNAL CALLS helps us to refine our understanding of these differences.

- We also learned that most accessors are readers, and there are only very few writers. Most of the writer methods are setters. This means that most of the attributes either are initialized when instances are created or are initialized and modified outside the class scope.
- If we consider the group of *collaborating attributes* taking into account only the direct accessors, we find that there are groups of two and three attributes, such as (wrapper, component), (bindings, window), (stack, composite), (policy, window), (source, bindings), (component, decorator, wrapper), (isEditing, labels, source). In the most of the cases, only 1 or 2 methods are accessing the groups of attributes by reading them. There are 3 attributes that are used alone in different methods. These facts reveal that the class is grouping several functionalities and could be split using the set of non-collaborating and collaborating attributes. This kind of hypothesis can be refined using the BEHAVIOURAL SKELETON view.
- In this specific case, we do not have any stateful core methods, which is not surprising as the class has a lot of attributes.

Validation with Scanner: The results for Scanner are completely different from those obtained for the other two classes. We find that we cannot partition the attributes into groups that are exclusively used by certain sets of methods. Instead, each method typically uses some subset of attributes that overlaps in arbitrary ways with those used by other methods. This means that every attribute offers some specific functionality that is complemented by the functionality offered by other attributes. None of the attributes have setters and getters, *i.e.*, the state is internal and it is not exported outside the scope of the class.

4.4.2 XRay View: EXTERNAL/INTERNAL CALLS

Description: Clusters methods according to their participation in internal or external invocations.

Used and Shown Groups: Interface Methods and Externally Used State.

Rationale: This view reveals the overall shape of the class in terms of its internal reuse of functionality. This is especially important for understanding framework classes that subclasses will extend. Interface methods, for example, are often generic template methods, and internal methods are often hook methods that should be overridden or extended by subclasses.

Validation with OrderedCollection: From the analysis of OrderedCollection, we obtain the following results:

- The *interface* is composed of 37 external methods. There are 22 methods (of those 37) that directly access attributes. Therefore the class OrderedCollection has a flat call-flow which means that there is little internal reuse of its own behaviour.
- The groups also reveal that on one hand we have methods such as add:, remove: that are part of the public class interface but are also used internally but they are not called *via a self-call*, and on the other hand we have pure, public methods such as changeSizeTo: and representBinaryOn:, meaning that they are not used at all in the class.

In VisualWorks [Vis03], the class OrderedCollection has 6 subclasses. However, each of these subclasses only *adds* extra behaviour and does not change the internal behaviour of the class. This confirms our expectations, since the absence of internal reuse of methods in OrderedCollection is also a sign that there is little behaviour to be reused or extended by subclasses.

Validation with UIBuilder: From the analysis of the class UIBuilder, the results show that:

- 89 of 124 UIBuilder methods are not invoked by the class itself.
- 35 methods define the internal behaviour of the class. This fact fits well with the intent of the Builder design pattern and the fact that UIBuilder offers not only a lot of functionality to build complex user interface but also offers several ways to query its internal state via methods such as componentAt:, listAt:, and menuAt:.
- We checked how the accessor methods identified by the STATE USAGE are classified as external and internal methods. For example, policy and decorator are external, as they allow the client of the builder to specify the look and feel policy used for the window. cacheWhileEditing is purely internal, as its name suggests. Note that this is a typical example how different views like STATE USAGE and EXTERNAL/INTERNAL CALLS complement each other in the process of understanding a class.

4.4.3 XRay View: BEHAVIOURAL SKELETON

Description: Clusters methods according to whether or not they work together with other methods defined in the class or whether or not they access the state of the class.

Used and Shown Groups: Collaborating Methods and Stateless Methods.

Rationale: Ideally an object should be cohesive. In reality, this is not always the case. For example user interface classes usually act as a glue between the domain objects and the widgets. The way methods form clusters of methods that work together indicates whether a class is cohesive or not [BDW98].

Validation with OrderedCollection: From our analysis we observe that we do not have much groups of *collaborating methods*. Let's see which are the characteristics we find in this class:

- If we have a look at the direct *collaborating methods*, i.e. methods that call directly other methods *via self* we find only groups of methods of no more than 4 methods that calls at most 2 methods. This means that there is no communication between groups of methods.
- There are several *calls** meaning that we have different groups of methods that call indirectly other methods. But in this specific case, we discover that the *indirect called methods* are subsets of the {lastIndex, changeCapacityTo:, increaseCapacity, size, firstIndex, makeRoomAtFirst }. In most of cases, these methods are *accessors* of the state of the class. We conclude then that most of the methods finally calls this *core* of methods, and as we said before, we confirmed that there is no communication between groups of methods.
- The two facts mentioned previously allow us to confirm what we saw in the STATE USAGE view that most of the methods access the attributes of the class, and what we saw in EXTERNAL/INTERNAL CALLS views that we have a flat method call-flow pattern for this class.
- One interesting feature of the groups *Collaborating Methods* is that the groups we found have a *naming* relationship, for example, in one group we have {add:, addAll:, addAllLast: } (a subset of the methods using in the *adding* interface of the class that have a common group of *called* methods).
- In the stateless methods, we identify two main groups:
  - methods inspectorClass and inspectorClasses that access three system global variables: OrderedCollectionInspector, BasicInspector and SequenceInspector [3].
  - methods that invoke methods, such as error:, inherited from the superclass.

[3] These global variables are used to identify the right class that can inspect an instance of OrderedCollection when debugging code

Validation with UIBuilder: In the EXTERNAL/INTERNAL CALLS view we see that UIBuilder has 35 methods that constitute the internal behaviour. We see that the call-flow is a complicated structure but the internal behavioral structure reveal the following issues:

- Each group of direct *collaborating methods* is reduced to, at most, 7 *called* methods calling *via self* 1 or 2 *accessors* methods of an attribute of the class;
- Groups of indirect *collaborating methods* are larger than the groups of direct ones, but finally in the most of the cases *called* methods are also *accessors* methods of an attribute of the class.
- From the two facts mentioned previously, we conclude that this confirms that most of the methods defined in the class are used as interface of the class and the state is exposed through the interface (as we saw in the EXTERNAL/INTERNAL CALLS view).
- Not as often as in the class OrderedCollection but in this case, we have also found that some methods in the groups were related by their names, *i.e.*, methods of the *opening* interface of a class.
- We have also identified 6 stateless methods. By inspecting the code, we see that these methods are not providing any specific functionality of the class and could be removed. By inspecting their code, we saw that one is catching an exception, one is returning a global variable, one does nothing, one is returning nil, one is returning the variable *self* and the last one is returning a boolean variable. By inspecting the code, we see that these methods are not so often used in the class so they are candidates to be removed.

Validation with Scanner: In this class, we identify a situation similar to that with OrderedCollection. The collaboration between the methods occurs in pairs, and there are no groups of methods collaborating with other groups. Since Scanner is a small class, it is not surprising that the internal collaborations are simple.

4.5 Application of the Approach: Analysis

In Chapter 3, we have discussed the approach in general terms. We have also mentioned different issues that the user must take into account when applying the approach. Here following we list the issues concerning XRay views on classes.

- *Choice of elements and properties.* Elements and properties are mapped directly from the meta-model: elements are attributes and methods, and properties are accesses to attributes and calls to methods.
- *Compact representation of data.* Supposing you have two methods m and n and one attribute a, if we have several calls to the method n or accesses to the attribute a in the method body of m, we just keep one representative of n and a related to the method m. So far, the number of calls and accesses appearing in the method m is not an analysis factor.
- *Limits of the XRay views.* In this high level view, we consider that all the possible XRay views are generated without considering inheritance relationships with other classes. We think that the limit with this specific set of properties is reached in this specific *high-level views* , and that more analysis in new classes will not generate new *XRay Views*.
- *Multiple FCA contexts.* In this case, we have used four lattices grouped in two. The first two ones are to analyze the state of the class directly and indirectly. The second ones are to analyze the invocations of the class directly and indirectly. We did not combine this information in a single lattice because we consider them to be completely different aspects of the class.
- *Unnecessary properties.* In some classes, the following properties *is Abstract, isStateless, isInterface* are discarded. This is normal because in any given class, it commonly occurs that all methods are concrete, or that all the method access the state, or that most methods are called inside the class .

- *Meaningless concepts.* We discarded concepts with a single method in the set of elements, because we were more focused on groups of methods (represented in the elements) collaborating with another group of methods (represented in the properties).

- *Mapping Partial Order of the Lattice.* We did not find a relationship that could be mapped to the partial order of the lattice.

4.6 Discussion

Use of FCA. The main advantage of using FCA is the possibility of specifying simple properties between the elements we need to analyze. Based on these simple properties, the FCA algorithm implemented in ConAn provides us with all the possible combinations of the elements with a set of properties that they have in common. This *mining* forms the groups we used to build the XRay views. The key issue is that we find groups of related entities that show us known and *unpredictable* relationships between the different elements. For example, we could detect (without using FCA) *all the methods accessing one specific instance variable* but FCA offers more than this information. We are able to identify all the possible groups of methods using groups of attributes and these groups are generated automatically by the FCA algorithm. However, not all the groups are meaningful for us. We need to perform post-processing to filter those groups that provide useful information about the class.

Overlap of Information. The results provided by different views will often overlap, as the views provide different perspectives of the same class using the same elements. For example, in the class UIBuilder the methods policy and decorator are Direct Accessors and Externally Used State. This means that these two methods appear in two XRay views: STATE USAGE and EXTERNAL/INTERNAL CALLS. This information redundancy is useful because we see that the XRay views are complementary ones, and reinforce each other.

Partial Information of the Concepts. Firstly when looking at the results of the groups to generate the XRay views, we see only partial information about the class. For example, when we analyze the group of *Collaborating Attributes* we see the group of methods that are accessing groups of attributes, but you can also have methods that are accessing each attribute, but not together. This means that in fact each group is determining a subgraph of the supergraph of all the calls/accesses of the class represented with the *Class Blueprint* of the analyzed class. This fact confirms that an analysis of each group in isolation provides incomplete results. We need to analyse the impact of the group in all the chosen class. To circumvent this problem we display the group under analysis in a *Class Blueprint* as shown in Figure 4.2, in this way we use the visualization of the complete call graph to provide a context to the group.

Opportunistic Code Reading. Using only XRay views, we are not able to analyze all the information about the class. To understand the class, the developer has to go iteratively between the views and reading the code. The XRay views show only the main skeleton of the class, but the opportunistic code reading is needed for an analysis.

Class functionality. We have seen that the results with *stateless* methods are not so interesting in the three analyzed classes. This does not mean that in all the classes we find the same situation, because it depends on the functionalities provided by the class. For example, in EJB [MH00] there is the idea of *stateless* and *stateful* session bean. An instance of a *stateful* bean is associated with one client. There is a one:one correspondence between session objects and the (stateful) instances of the session bean class. The EJB container always delegates requests from a given client to the same session bean instance. However, the stateless session objects do not retain any client-specific state between invocations. The EJB container maintains pool of instances of the session bean class and delegates clients' requests to any available instance. Then, we see that with other classes we can discover other system features based on the use (or not) of the state.

4.7 Related Work

Among the applications for understanding object-oriented systems at the class level using FCA, we only found Dekel work [Dek03]. Dekel uses FCA to visualize the structure of the class in Java and to select an effective order for reading the methods. He calculates all the accesseses to fields that each method makes. In our approach, we also calculate the access to the instance variables of a class, but in contrast to our approach, he does not provide information about the interaction between the methods, nor does it reveal whether a method accesses a combination of fields directly, by accessing their values, or indirectly, by invoking methods that access them directly. To detect all the mentioned features, he superimposes the *method call-graph* onto the concept lattice and obtains a *embedded call-graph*.

There is also some relevant work to support the understanding of object-oriented systems at the class level that is not based on FCA. GraphTrace [KG88] visualizes concurrent animated views to understand the way a system behaves. Program Explorer [LN95] uses both dynamic and static information that the reengineer can query and visualize function invocation, object instantiation, and attribute access via simple graphs. The views show class and instance relationships (usually focused on a particular instance or class), and short method-invocation histories. Using basic graph visualizations to represent various relationships, Mendelzon and Sametinger [MS95] show that they can express metrics, verify constraints, and identify design patterns. Cross *et al.*, in the context of procedural languages, have proposed and investigated new control structure diagrams to support the reading of the applications' control flow [CIMH98]. Lanza and Ducasse have proposed *Class Blueprints*, which are structured call flows enriched with semantical information and metrics [LD01]. Finally, program slicing [GL91] is also used to support the understanding of programs. Based on slices, CodeSurfer [AT01] supports understanding by using hypertext facilities. In all these approaches, the visualization of call graph of a class is the main source of the analysis, they combine static and dynamic information of a system, and they focus partially on a class. In our approach, the visualization is used as a secondary tool to analyze the class based on the generated *XRay Views*, and we restrict ourselves to static information of a system and we analyze the complete class.

4.8 Conclusions

This section summarizes the *views* on understanding a class, a discussion about the approach, the answers to the research questions proposed in Chapter 2 and future work.

4.8.1 Summary

In this chapter we have introduced one approach based on FCA to help in the understanding of object-oriented classes. The identified concepts show dependencies between groups of methods and attributes of a single class. Using them, we have defined a number of useful XRay views which correspond to related groups that expose specific aspects of a class, and they are particularly useful for understanding the behaviour of a class. We have validated the technique by applying it to a number of Smalltalk classes using ConAn, a tool we have developed to automatically generate the groups that compose the XRay views.

In our first experiences we can observe the following:

- each XRay view has a clear focus, and identifies a group of methods exhibiting some key properties,
- the views do not stand on their own, but complement and reinforce each other, because they analyze the class from different viewpoints but with the same elements,
- although the generation of groups and the views is fully automatic, their interpretation entails iterative application of views and opportunistic code reading,
- the user has to know the semantics of the groups he is using so a learning phase is necessary, and
- the current approach does not take inheritance into account, which can be an impediment to understanding.

4.8.2 Research Questions

Based on the approach, we are able to answer some of the research questions proposed in Chapter 2:

- *How does the technique help in understanding the inner workings of a class ?*

  In this approach, we have seen that FCA is an useful technique in understanding a class. Using the attributes and methods as FCA elements and mapping the accesses and invocations as FCA properties, we have built three different XRay views that help us to focus on different characteristics implemented in a class. The three XRay views do not stand on their own, but complement and reinforce each other. The interesting characteristic of the approach is the possibility to discover known and unknown groups of attributes and methods revealing *implicit* contracts. These contracts are shown in the different XRay views.

- *Is there a limited number of XRay views in a class ?*

  We consider that the number of XRay views is not limited. We have defined only three XRay views analyzing the class as a sole development unit. If we consider the inheritance relationships with its subclasses and superclasses, and how the clients classes use the class, the class will reveal new *implicit* contracts and also new XRay views will appear.

- *Does FCA discover new dependencies in the class ?*

  Our FCA-based methodology applied in understanding a class discovered new dependencies in the class. Two concrete examples are groups of methods using groups of attributes, and also indirect accesses and invocations in the class. These two cases are not trivial to detect in the classic code-reading technique.

4.8.3 Future Work

Our next steps consists of:

- Use of repeated information (we would not apply the issue of *Compact representation of data*) and measurement the frequency of accesses of attributes and calls to methods in the concepts of the lattice.
- Extraction of new kinds of views considering possible relationships in a context of a class hierarchy and also the possible relationships with other class -not necessarily presented in the class hierarchies.
- Controlled experiment with developers to evaluate the advantages and drawbacks of the application of *XRay Views*.
- Refinement of the used properties and definition of new complex ones.
- Definition of filters to remove more non-meaningful information from the lattice.

Chapter 5

Hierarchy Schemas: Discovering Unanticipated Dependencies in Class Hierarchies

Inheritance is the cornerstone of object-oriented development, enabling conceptual modeling, subtype polymorphism and software reuse. But inheritance can be used in subtle ways that make complex systems hard to understand and extend [DDN02]. In particular, a developer making changes or extensions to an object-oriented system must understand the implicit contracts and dependencies between a class and its subclasses, or risk that seemingly innocuous changes break these contracts [SLMD96]. In this chapter we focus on the second abstraction level of a system: a class hierarchy. We develop a methodology focused on identifying undocumented hierarchical dependencies in a hierarchy only taking into account the existing collaborations between classes and subclasses. Our analysis is based on the internal structure of each class, and focus on how the class define and use local behavior and state, and define and use superclass behavior and state. We provide a catalog of *Hierarchy Schemas*, each composed of a set of dependencies over methods and attributes in a class hierarchy. Each of these schemas shows the *hidden contracts* of the classes in the class hierarchy. These schemas identify different design features of the class hierarchy: *Classical Schemas* represent common styles that are used to build and extend a class hierarchy, *"Bad Smell" Schemas* represent doutbul design decisions that should be completely changed, and *Irregularities Schemas* represent often implementation that could be improved using minimal changes.

5.1 Problems in Understanding Class Hierarchies

Class hierarchies are another cornerstone of the object-oriented development, enabling conceptual modelling, subtype polymorphism and software reuse [DDN02]. However, several building mechanisms make them difficult to understand. Let's mention some of them:

- One programming practice is the use of subtyping or subclassing. The semantics associated with inheritance may be inconsistent within a single hierarchy. The *subclassing* pushes reuse to its limits and makes understanding more difficult as the concepts are not related and understanding a subclass requires to understand how the *implementation* of the superclass is reused. It forces the developer to cancel inherited methods or to invoke hidden methods with different names to literally jump over methods implemented in the class [KST96].
- The presence of late-binding leads to "yoyo effects" when walking through a hierarchy and trying to follow the call-flow [WH92].
- Classes define state and the methods that act on this state. It is important to understand how the state

is accessed, presented, if at all, to the class's clients, and how subclasses access this state.

5.2 Goals of Hierarchy Schemas

The identified dependency schemas on class hierarchies help us to answer such questions as:

- Which classes define and use (or not) their own state and behavior
- Which classes use the state defined in their superclasses
- Which classes use *template and hook methods* and define behavior for their subclasses
- Which classes reuse or extend (or not) the behavior of their superclasses
- Which classes cancel the behavior of their superclasses

Furthermore, the approach is entirely neutral. Uncovered dependency schemas may correspond either to well-known best practice in object-oriented design, or they may be signs of degenerated design. Once dependency schemas are classified, they are a good basis for identifying which parts of a system are in need of repair. The approach thus provides us not only with a *global* view of the system and which kinds of dependencies and practices occur, but it also provides *detailed* information about how specific classes are related to others in their hierarchy, and how that hierarchy can be modified and extended.

Structure of the Chapter. This chapter is structured as follows: In Section 5.3 we present our mapping of object-oriented dependencies to the framework of FCA. In Section 5.4 we provide an overview of the results obtained by applying our approach to the *Smalltalk Collection* hierarchy and by showing how SortedCollection fits into this hierarchy. Section 5.5 analyzes several issues (introduced in Chapter 3) related to the application of the approach in class hierarchies.Section 5.6 provides a brief discussion of various technical issues. Section 5.7 presents some related work. Finally, we conclude and outline about the future work.

5.3 Formal Concept Analysis in Analyzing Class Hierarchies

In order to apply FCA to detect dependencies schemas, we must cast models of OO software systems in terms of an FCA context, that is, we must define the elements and properties of interest. We will first describe this context, and then show how recurring combinations of properties lead to the dependency schemas of interest.

5.3.1 Elements and Properties of Classes

We choose as elements the *invocations* of methods via a *self* or a *super* send and *accesses* to the attributes of a class. We choose as properties of invocations whether the call is a *self* or a *super* send and the relationships between the class that *defines* and the one that *invokes* the methods. In case of accesses, we are interested in the relationships between the class that defines the attribute and the one that accesses it.

To this end, we define the following predicates with the obvious meanings, where m is a method, a an attribute, i is an invoked method or accessed attribute, and C, C_1, C_2 are classes:

- *i is an invoked method in C*
- *i is an accessed attribute in C*
- *C invokes i via self*
- *C invokes i via super*

- C *defines* i
- C_1 *is ancestor of* C_2
- C_1 *is descendant of* C_2
- m *is abstract* in C
- m *is concrete* in C
- m *is cancelled* in C

We now combine these predicates in the obvious way to obtain the following list of 15 properties:

- i *accesses* { *local state, state in Ancestor* C_1, *state in Descendant* C_1 } (3 properties)
- i *is defined* { *locally, in ancestor* C_1 *of* C , *in descendant* C_1 *of* C } (3 properties)
- i { *is abstract, is concrete, is cancelled* } { *locally, in ancestor* C_1 *of x* C, *in descendant* C_1 *of* C } ($3 \times 3 = 9$ properties)

For example, the property i *accesses local state* is trivially obtained by combining i *is an accessed attribute in* C and C *defines* i.

We also directly adopt the following 2 predicates as properties, leading to 17 properties considered in total:

- C *invokes* i *via self*
- C *invokes* i *via super*

5.3.2 Interpretation of the Properties in Concepts

By applying FCA to this context, we obtain certain recurring sets of properties as "concepts". Certain of these concepts correspond to interesting dependency schemas, as reported in Section 5.4. Let us briefly consider two examples.

The schema *Reuse of Superclass Behavior* is composed of the following properties:

- C *invokes* i *via self*: {copyEmpty, insert:before:, reverseDo:, asArray, isEmpty, notFoundError} are *self-called* in SortedCollection
- i *is concrete in ancestor* C_1 *of* C : {copyEmpty, insert:before:, isEmpty} has *concrete behavior in ancestor* OrderedCollection; and {reverseDo:, asArray} has *concrete behavior in ancestor* SequenceableCollection; and {notFoundError} has *concrete behavior in ancestor* Collection

Within the *"Bad Smell"* category, we have the schema *Broken* super *send Chain* and it is composed of the following groups:

- C *invokes* i *via super*: {representBinaryOn:, =} are *super-called* in SortedCollection
- i *is concrete locally*: {representBinaryOn:, =} has *concrete behavior in ancestor* SortedCollection
- i *is concrete in ancestor* C_1 *of* C : {representBinaryOn:, =} has *concrete behavior in ancestor* SequenceableCollection
- i *is concrete in descendant* C_1 *of* C : {representBinaryOn:, =} has *concrete behavior in descendant* SortedCollectionWithPolicy

5.4 Detected Dependency Schemas

We present here the results of our analysis of the Smalltalk Collection hierarchy. This hierarchy is especially interesting because (i) it is an essential part of the Smalltalk system, and (ii) it makes heavy use of subclassing for a variety of purposes. It is an industrial quality class hierarchy that has evolved over 15 years, and has been studied by other researchers [GMM+98] [Coo92]. It has also influenced the design of current C++ and Java collection hierarchies. The Smalltalk Collection hierarchy is composed of 104 classes distributed over 8 levels of inheritance. There are 2162 defined methods in all the classes, with 3117 invocations of these methods within the hierarchy and 1146 accesses to the state of the classes defined in the hierarchy.

We will first provide a *global* overview of the schemas discovered, and then we will focus on the role of the class SortedCollection within the collection.

5.4.1 Global View on Collection Hierarchy

By applying FCA to the Collection hierarchy, we discovered 451 instances of 16 different dependency schemas. We were then able to manually categorize these into three groups: *Classical*, *"Bad Smell"* and *Irregularities*.

- *Classical* Schemas represent *common* idioms/styles that are used to build and extend a class hierarchy, *i.e., best practices*.
- *"Bad Smell"* Schemas represent doubtful designs decisions used to build the hierarchy. They are frequently a sign that some parts should be completely changed or even rewritten from scratch.
- *Irregularities* Schemas represent *irregular* situations used to build the hierarchy. Often the implementation can be improved using minimal changes. They are less serious than *"Bad Smell"* schemas.

Table 5.1 provides an overview of the three groups, together with the total number of detected instances of each schema.

***Classical*: Local Direct State Access.** This schema identifies classes that define and use their own state directly (using or not the accessors). In Collection hierarchy, there are 55 classes contained in this schema. Most of the classes are leaves in the class hierarchy represented as a tree, and it shows that this hierarchy is built based on *subclassing* principle, because each class is extending behavior inherited from the superclasses and providing specific functionality. Only in the subhierarchies starting from String and WeakDictionary have no leaves classes that fullfil this form, meaning that eventually these classes either use state of the superclasses or only extend the behavior of the superclasses without extending the state of the superclasses.

***"Bad Smell"*: Ancestor Direct State Access.** This schema identifies classes that access (read or modify the values of) the state of an ancestor class without using the accessors defined in the ancestor classes. We identified 19 classes that are part of the subhierarchies determined by GeneralNameSpace, Dictionary, OrderedCollection, LinkedList. In most of the cases, the classes are accessing state of the immediate superclass, but in the subhierarchy of OrderedCollection we detected several classes that access state of ancestors higher up in the chain of their superclasses. This is a not good coding practice since it introduces an unnecessary dependency on the internal representation of ancestor classes, and thereby violates encapsulation. Figure 5.1 illustrates this schema.

***"Bad Smell"*: Cancelled Local or Inherited Behavior.** This schema identifies concrete or local inherited methods that are invoked via a *self* send in a class or its superclasses but are then cancelled in subclasses. Method cancellation is a sign that inheritance is being applied purely for purposes of code

Name	Description	Nr.
Classical		
Local Direct State Access	Identifies methods that directly access the class state. *Variations:* using or not using accessors.	72
Local Behavior	Identifies methods defined and used in the class and that are not overridden in the subclasses. Often represent internal class behavior.	69
Template And Hook	Identifies methods that define template and hook methods. *Variations:* default hooks are abstract or represent a default behavior.	17
Redefined Concrete Behavior	Identifies concrete inherited methods that are redefined in the class or in the subclasses.	43
Extended Concrete Behavior	Identifies concrete inherited methods that are extended in the class (only *super* send).	37
Reuse of Superclass Behavior	Identifies concrete methods that invoke superclass methods by *self* or *super* sends. *Variation:* method that invokes super method of the same name.	111
Local Behavior overridden in Subclasses	Identifies methods that are overridden in subclasses	29
Abstract and Concrete Chain	Identifies an abstract method, and a chain of subclasses that override it with a concrete implementation.	10
Bad Smells		
Ancestor Direct State Access	Identifies methods that directly access the state of an ancestor, bypassing any accessors.	19
Cancelled Local Behavior but Superclass Reuse	Identifies concrete inherited methods whose behavior is *cancelled* in the class but whose corresponding superclass behavior is invoked *i.e.*, via a *super* send from a different method. This workaround is a common sign of difficulty improperly factoring out common behaviour.	1
Abstracting Concrete Methods	Identifies abstract methods overriding concrete ones.	8
Cancelled Local or Inherited Behavior	Identifies concrete or local inherited methods that are invoked *i.e.*, via *self* send in a class or its superclasses, but are cancelled in subclasses. Method cancellation is a sign of inheritance for code reuse without regard for subtyping.	6
Broken *super* send Chain	Identifies methods that are extended (*i.e.*, via a *super* send) at some point in the hierarchy, but are then simply overridden lower in the hierarchy. This can be the sign of a broken subclassing contract.	7
Irregularities		
Inherited and Local Invocations	Identifies methods that are invoked by both *self* and *super* sends within the same class. This may be a problem if the super sends are invoked from a method with a different name.	15
Unused Local Behavior but Superclass Reuse	Identifies concrete inherited methods whose behavior is *overridden* but unused in the class, and whose corresponding superclass behavior is invoked *i.e.*, via a *super* send from a different method.	3
Accessor Redefinition	Identifies methods that are accessors in a class but are redefined in the subclass as non-accessor methods.	4

Table 5.1: Commonly Identified Schemas.

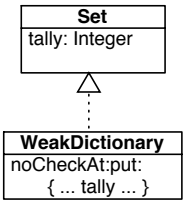

Figure 5.1: Ancestor Direct State Access.

reuse, without regard for subtyping. Since methods of the superclass calling the cancelled methods can still be called on the cancelling class, this may lead to runtime errors. In the Collection hierarchy it occurs in the subhierarchies of SequenceableCollection and OrderedCollection. Figure 5.2 illustrates this schema.

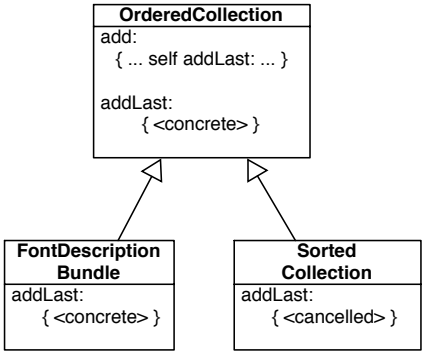

Figure 5.2: Cancelled Inherited Behavior.

***Irregularities*: Inherited and Local Invocations.** This schema shows methods that are invoked by both *self* and *super* sends within the same class. Initially this schema is a *good practice* coding (as shown in Figure 5.3), but a problem occurs when the *super* sends are invoked from a method with a different name. This special case of the schema occurs in the classes LinkedOrderedCollection, LinkedWeakAssociationDictionary and XMainChangeSet. All these classes have a special form: the class overrides a method m and m invokes a method named own-m via *self* send, and this last method calls m via a *super* send implemented in the superclass. Figure 5.4 illustrates this schema. This is an *irregular* case of the schema Redefined Concrete Behavior because the class is overriding the superclass behavior but is indirectly using the superclass behavior.

5.4.2 "Class-Based" View on SortedCollection

With the *global view* we analyze a class hierarchy, but our approach helps also us to analyze how a class is built in the context of its superclasses and subclasses.

We chose to analyze the class SortedCollection (a subclass of OrderedCollection). A SortedCollection is an ordered collection of elements, using a sorting function for the elements. The class has one attribute sortBlock which holds the sorting function; has one class variable (static variable in Java) DefaultSortBlock that holds the default sorting function. As a subclass of OrderedCollection, it inherits two instance variables firstIndex and lastIndex and an indexed variable objects. Regarding its methods, it defines 10 methods and overrides 19 methods from the 403 inherited.

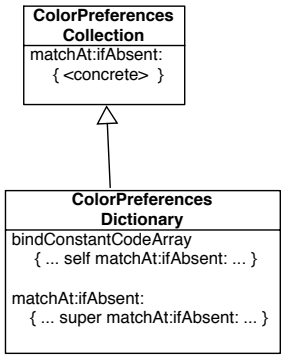

Figure 5.3: Inherited and Local Invocations - Case 1

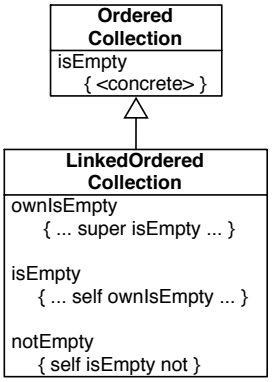

Figure 5.4: Inherited and Local Invocations - Case 2

In this class we identify 12 different schemas that involves this class.

Within the *Classical* category we report 1 case.

- **Reuse of Superclass Behavior:** This schema shows us that the class SortedCollection calls via *self* the methods copyEmpty, insert:before:, reverseDo:, asArray, isEmpty, notFoundError and they are not defined in the class but different superclasses define their behavior. Specifically, we see that the methods copyEmpty, insert:before: and isEmpty are defined in the class OrderedCollection, reverseDo: and asArray are defined in the class SequenceableCollection; and notFoundError is captured in the class Collection. Thus, we see which are the superclasses that determine the behavior of the class. Figure 5.5 illustrates this schema. In Figure 5.6 we see how our visual tool shows this schema.

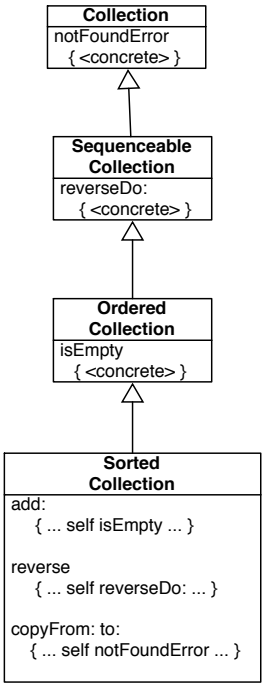

Figure 5.5: Reuse of Superclass Behavior.

Within *"Bad Smell"*, we report 2 cases:

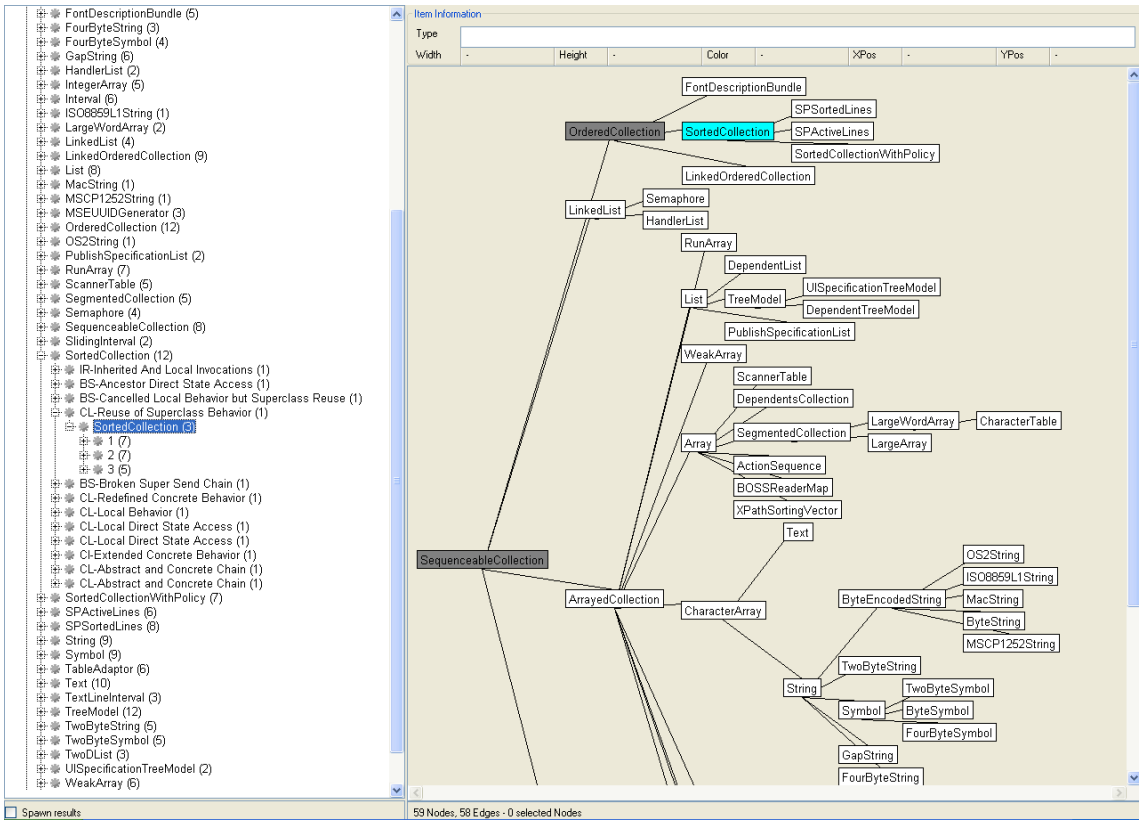

Figure 5.6: Dependency Schemas identified in SortedCollection. All the classes are listed in the left pane. The schema Reuse of Superclass Behavior is visualized in the right pane identifying the classes related to SortedCollection in gray

- **Broken *super* send Chain:** This schema identifies methods that are extended (*i.e.*, performing a *super* send) in a class but redefined in their subclasses without calling the overridden behavior, hence giving the impression to break the original extension logic. In SortedCollection the methods = and representBinaryOn: are invoking superclass hidden methods. But the definition of these methods in the subclass SortedCollectionWithPolicy does not invoke the method defined in SortedCollection. Such a behavior can lead to unexpected results when the classes are extended without a deep knowledge of them. Figure 5.8 illustrates this schema.
- **Cancelled Local Behavior but Superclass Reuse:** This schema shows that the method addLast: is called *via a super send* and this method is defined in the immediate superclass OrderedCollection, meaning that the class is reusing the behavior of the superclass. But this method is also implemented in the class SortedCollection but the behavior is *cancelled*. Although it is not a good practice, it seems a normal situation because the elements in a *sorted collection* cannot be added in the end of the collection, but in a predefined position defined by the sorting function of the class. As we said previously, this is a case where the inheritance is used as code reuse without regarding *subtyping*. Specifically, this means that SortedCollection is a kind of OrderedCollection but not all the inherited methods can be applied. Figure 5.7 illustrates this schema.

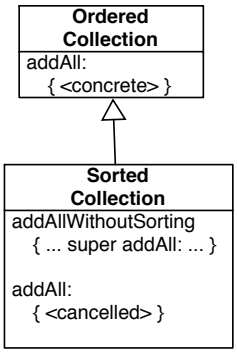

Figure 5.7: Cancelled Local Behavior and Behavior Reuse of Superclasses

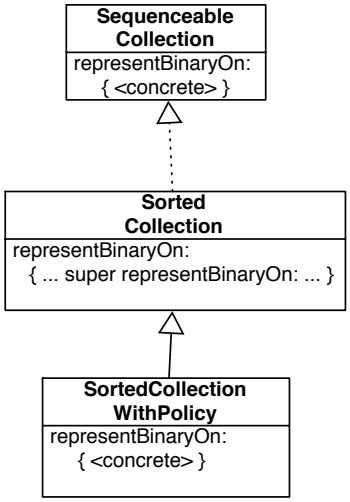

Figure 5.8: Broken *super* send Chain

Within the *Irregularities*, we only found one case:

- **Inherited and Local Invocations:** This schema shows that the method copyEmpty is used with *self* sends and *super* sends in the class *SortedCollection*. It is implemented in the class itself, has an implementation in the superclass Collection and an implementation in the subclass SortedCollection-WithPolicy. When checking the code, we see that the most of the calls are *self* sends and in the method called copyEmpty, we have a *super* send to a method with the same name. This means that, in spite of a local implementation of copyEmpty, finally the behavior of this method is determined by the superclasses, showing a heavy reuse of the superclass code. Figure 5.9 illustrates this schema.

The identified schemas in our approach provides another view on the class. They present some anticipated dependencies between the methods of the classes and their relationships in the hierarchy. Our experience confirms to us that the Collection hierarchy is a rich but difficult to extend hierarchy since it is based on a heavy use of subclassing and aggressive code sharing. It relies on some internal knowledge and often contains coding manner that leads to fragile design.

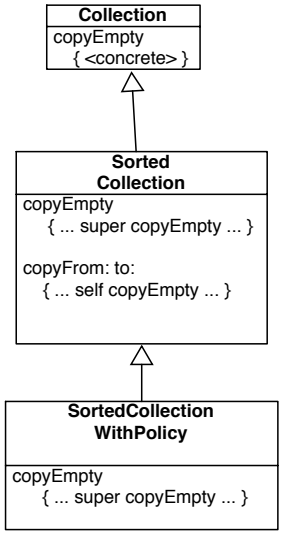

Figure 5.9: Inherited and Local Invocations.

5.5 Application of the Approach: Analysis

In Chapter 3, we have discussed the approach in general terms. We have also mentioned different issues that the user must take into account when applying the approach. Here following we list the issues concerning *Hierarchy Schemas* on class hierarchies.

- *Choice of Elements and Properties.* We map the attributes accesses and methods calls directly from the metamodel. The choice of properties requires some analysis, because we need to cover the different possible inheritance relationships of the elements. The properties C *invokes* i *via self* and C *invokes* i *via super* are mapped directly from the metamodel the rest of the properties are calculated based on the relationships expressed in the metamodel.
- *Compact representation of data.* Supposing you have one method m and one attribute a in a class C, if we have several calls to the method n or accesses to the attribute a in several methods of the class C, we just keep one representative of n and a related to the class C.
- *Use negative properties.* We define three negative properties because they help us to complement the information of the elements considering the three inheritance relationships used in the approach: *local*, *ancestor* and *descendant* definitions.
- *Single context.* In this case, we just use only one lattice, because we analyze only one aspect of classes: inheritance relationships.
- *Meaningless concepts.* All the meaningful concepts must show either a positive or negative information about the 3 relationships: *local*, *ancestor* and *descendant*, and have at least one property of the set $\{C$ *invokes* i *via self*, C *invokes* i *via super* $\}$. The rest of the concepts are discarded.
- *Mapping Partial Order of the Lattice.* We did not find a relationship that could be mapped to the partial order of the lattice.
- *Equivalent Concepts.* We have said before that each concept is a candidate to be a schema. In this case study, we found that several concepts could represent the same schema, meaning that a schema can be expressed with different group of properties.

5.6 Discussion

Performance of the Algorithm: In Chapter 3, we saw that we need to map the model entities (in our specific case, the invocations and accesses) to *FCA elements* and build different properties based on them. Due to a limitation imposed by FCA algorithm in performance measurements [KO01], we reduce the amount of FCA elements to compute the concepts and the lattice without losing information about the class hierarchies. Thus, if we have several invocations of the same method or several accesses to attributes in the same class, we keep only one invocation or one access per class as a representative, and we reduce dramatically the number of FCA elements used by ConAn, and reduces the computing time from around 1 hour to 10 min compared to the approach presented in a previous work [Aré03].

Analysis of State and Behavior: There are two main differences with regard to our previous work [Aré03]. First, in the current approach we take dependencies to state into account, whereas our earlier work considered only behaviour. This yields more concepts, and hence more schemas of interest than when only behaviour is considered. Secondly, we are able to categorize schemas into those that represent *good*, *irregular* and *bad* design decisions in the class hierarchies.

Partial Usage of Lattice: We pointed out that once the concepts and the corresponding lattice are built, each concept represents a group of invocations and accesses that relate a group of classes. But not all the concepts are relevant, and we keep only the meaningful concepts. There are main three points worth mentioning. Firstly, if we analyze the position of those concepts in the lattice, we see that most of them are located in the lower part in the lattice, and we filter out the concepts located in the middle and upper part of the lattice. This is because the concepts in the lower part of the lattice contain more properties (inversely, few elements with those commonalities) than concepts higher up in the lattice (inversely, more elements with less commonalities). Thus, the lower concepts provides more "interesting" information (based on the combination of properties) and allow us to map them to non-trivial schemas of classes in a hierarchy. Secondly, we only use 64 of 174 concepts in total, meaning that the 1/3 of the lattice is used. Finally, we must note that, in this particular application of FCA, we do not use the *partial order* of the concepts in the lattice. This means that we do not exploit the possible relationships between the schemas (mapped from the concepts).

Mapping from Concepts to Schemas: Of the 64 concepts we identify as "interesting", we derive 16 dependency schemas. This means that in most of cases, a schema is represented in several concepts, meaning that a schema can be described by different combinations of properties. But the policy of mapping is arbitrary so far, meaning that when we interpret the contents of the concepts, we decide which are concepts corresponding to the different schemas. For example, the schema *Local Direct State Access* is represented in 5 concepts because each concept shows different ways that the state of the class is accessed. On the other hand, the schema *Local Behavior* is represented by just one concept. In other cases, one schema could represent a *good* or an *irregular* design practice. In this specific case, we see that the schema *Inherited and Local Invocations* is *irregular* only when the *super* sends are invoked from a method with a different name.

"Non-invoked" Methods: Our approach is limited to analyzing methods and attributes that are effectively used in the context of the class hierarchy. If there are methods that are defined in any class but are not invoked in the class itself or in any subclasses or in any superclasses, those methods are not included in our analysis. Clearly, we lose some information about the classes in the hierarchy, because we only concentrate on usage of behavior and state of the class.

FCA vs. Logic Engine: One of the main results of this approach is a *catalog* of schemas to characterize a class hierarchy. As we see in Section 5.3, each schema is the interpretation of a *conjunction* of properties in the concepts. Then, each schema can be expressed as a logic predicate (mapped from the properties) and a logic engine can be run in a class hierarchy to identify the occurrences of the different schemas. Thus, the main difference between the use of FCA and a logic engine is that, in the former case, we do not know in advance which are the possible schemas occurring in the class hierarchies, and consequently we do not know the combination of properties that characterize them. FCA helps us mainly to discover *unpredictable* schemas introduced in a class hierarchy. In the case of the use of a logic approach, we need

to know which are the different properties that characterize a schema. We consider that both approaches are complementary ones, because the catalog of schemas can be complete after the analysis of several class hierarchies, and in that case, the application of logic engine is most suitable than FCA.

5.7 Related Work

Various researchers have explored techniques to support the understanding and evolution of class hierarchies. We briefly survey a small selection.

Steyaert et al. introduce the concept of *reuse contracts* to capture the specialization interface between a class and its subclasses [SLMD96]. Reuse Contracts can be used to identify conflicts during the evolution of a hierarchy.

Program Explorer [LN95] enables a software engineer to query and visualize both dynamic and static information via simple graphs to understand and verify hypotheses about function invocations, object instantiation and attribute accesses. Using basic graph visualizations to represent various relationships, Mendelzon and Sametinger [MS95] show that they can express metrics, constraints verification, and design schema identification.

However, few approaches have focused on understanding complete class hierarchies. Ducasse and Lanza [LD01] introduce the notion of a *Class Blueprint*, a semantically augmented visualization of the internal structure of a class, which displays an enriched call-graph with a semantics-based layout. With this approach the reader can perform the analysis and browse the code to validate his hypotheses. Keller et al. [SRMK99] focus on the identification of hook and template methods.

Several researchers have also applied FCA to the problem of understanding object-oriented software. Godin and Mili [GMM+98] uses FCA to maintain, understand and detect inconsistencies in the Smalltalk Collection hierarchy. They show how Cook's [Coo92] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. In C++, Snelting and Tip [ST98] analysed a class hierarchy making the relationship between class members and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. As a result, they propose a new class hierarchy that is behaviorally equivalent to the original one. Similarly, Huchard [HDL00] and Leblanc [Leb00] applied concept analysis to improve the generalization/specialization of classes in a hierarchy. Based on this approach, Roume [Rou02] and Dao et. al. [DHL+02] defined metrics to measure the impact of refactorings to the class hierarchies.

All the above approaches only take into account which selectors are implemented by which classes. They do not consider behavioral information (*i.e.*, based on *self* and *super* sends) or usage of the state defined in the classes. As shown in this thesis, this information in static analysis helps us to identify different *behavioral* and *state dependency schemas*. With these schemas, we evaluate the reuse of the methods and state defined in the classes, and we discover different *design* decisions used in building the class hierarchies.

5.8 Conclusions

This section summarizes this approach with a summary, a discussion, the answers to the research questions proposed in Chapter 2 and future work.

5.8.1 Summary

In this chapter, we show how the automatic generation of schemas using FCA helps us to discover different implicit and undocumented dependencies in class hierarchies in terms of the behavior and state usage. The categorization of these schemas into *good*, *irregular* and *bad* design decisions helps us to:

- Generate the first mental model of a hierarchy.

- Localize where different irregularities or problems occur in the implementation of a class hierarchy.
- Identify which are the main constraints that a specific class has in the context of a hierarchy. When a developer wants to use or extend a class, he needs to understand which are the different dependencies a class has regarding its subclasses and superclasses. These dependencies include if the class is overriding or reusing the behavior inherited from the superclasses or if the class has template methods and hooks that its subclasses should implement.

The large number of concepts to be mapped as schemas was the main drawback in the first application of this approach. In spite of that, we believe that the catalog of schemas shows us useful information about class hierarchy specially in cases such as Smalltalk Collection that have evolved with different building principles and show different coding styles in the complete hierarchy.

5.8.2 Research Questions

Based on the approach, we are able to answer some of the research questions proposed in Chapter 2:

- *How does the technique help in discovering schemas introduced in a class hierarchy ?*

  In this approach, we have seen that FCA can also be used in analyzing class hierarchies (our second abstraction level in a system). In this case, we use accesses of attributes and method calls in a class hierarchy as FCA elements, and we define properties based on where the attributes and methods are defined. With this information we build a catalog of recurring situations named as *schemas*. Each schema is classified as *good*, *irregular* or *bad smell* according to the kind of design decision they represent. These schemas clearly help the developer to identify good coding situations, pieces of code that could be improved with minimal changes and cases where a design from scratch is needed.
- *Do the schemas show new dependencies in the class hierarchy ?*

  The *irregular* and *bad smell* schemas reveal us that the class hierarchy presents new unexpected dependencies. In those situations, we think that FCA is an useful technique. *Classical* design decisions in a class hierarchy are expected because they show known and often used building mechanisms. Meanwhile, *Irregularities* and *"Bad Smell"* schemas show that the code was changed without taking into account existing contracts between the classes. These *hidden* contract are difficult to grasp when the developer is not aware of them. Any change break the existing collaborations in these situations.
- *Can FCA help to identify situations where the developer could apply reverse engineering tasks ?*

  This question is related to the second one. *Irregularities* and *"Bad Smell"* schemas represent in all the identified cases opportunities to apply refactorings in the code. We are only able to identify those situations. So far, we did not evaluate if the needed refactoring are costly in terms of changed code.

5.8.3 Future Work

Our next steps consist of:

- Application of the approach to other class hierarchies to check if the catalog of schemas identified thus far covers all the interesting possible cases or if we discover new cases of implicit contracts.
- Extended analysis comprising methods that are invoked and those that are not invoked but are declared in the classes. This kind of approach can measure how much information defined in the class hierarchy is used or not.
- Refinement of properties to get a mapping 1 to 1 from concept to schema, and thus, reduce the complexity of the lattice in terms of number of concepts.
- Analysis of relationships given by the *partial order* between the different schemas -mapped from the concepts in the lattice- to find a possible mapping in terms of software reengineering.

Chapter 6

Collaboration Patterns: Detecting Implicit Dependencies in Applications

One of the key difficulties faced by developers who must maintain and extend complex software systems, is to identify what are the *implicit contracts* in the system. Such contracts are typically manifested as recurring patterns of software artifacts, which may represent design patterns, architectural constraints, or simply idioms and conventions adopted for the project. We introduce the term *Collaboration Pattern* to cover all these cases.

In most applications, the implicit contracts may be recovered by recognizing occurrences of collaboration patterns in the source code [CC92] [NSW+02]. However this task is anything but trivial in medium-sized to large applications. In most cases, the documentation of the systems is out-of-date, and the information we are looking for is not explicit in the code [DDN02], [SLMD96], [LRP95].

In this chapter, we explore an approach for detecting collaboration patterns that refines and extends that proposed by Tonella and Antoniol [TA99] for detecting classical design patterns. We focus on the third abstraction level of a system: a complete application. We take the source code of an object-oriented application as our main information source and extract structural relationships between classes. We then apply Formal Concept Analysis (FCA) to identify recurring "concepts" (*i.e.*, patterns) in the software.

6.1 Goals of Collaboration Patterns

Our approach consists of:

- improvements to the pattern detection algorithm used by Tonella and Antoniol to avoid redundancy in the representation of structural relationships and improve the time performance of calculating concepts,
- generalization of the technique to a language-independent approach,
- the introduction of a filtering phase to narrow the scope of candidate patterns.

Based on the results from our experiments, the main contributions of this chapter are:

- the detection of both classical and non-classical patterns in various applications using simple structural relationships between classes. We are not limited to known design patterns but can detect *any* recurring collaboration between classes in the analyzed applications.
- the possibility of establishing relationships, called *pattern neighborhoods*, over detected patterns. With the *neighbors* of the patterns, we can detect either missing relationships between classes needed

to complete a pattern, or excess relationships between classes that extend a *pattern*. We can also analyze the connections of the identified patterns with the classes implemented in the analyzed application.

- the incremental construction of a pattern library to match candidates against known design patterns and detected patterns after each case study.

For the sake of conciseness, we use the term *pattern* to refer to *collaboration patterns* in the rest of the chapter.

Structure of the Chapter. The chapter is structured as follows: We describe in detail the steps of the pattern detection approach in section 6.2. We describe and evaluate our experimental results in sections 6.3. Section 6.4 analyzes several issues (introduced in Chapter 3) related to the application of the approach in analyzing an application. Section 6.5 we provide some discussion points based on the experiments of generating collaboration patterns. Section 6.6 presents some related work to detection of patterns using other techniques. And final Section 6.7 we conclude the chapter and outline some future work.

6.2 Formal Concept Analysis in Analyzing an Application

In this section we outline in detail the three processing steps we have to deal with to get the concepts meaningful enough that will interpreted as *Collaboration Patterns*.

6.2.1 FCA Mapping: Setup of the Formal Context

In order to use FCA, we need to define the elements and properties of a context $\mathcal{C}$. The **elements** $\mathcal{E}_o$ are tuples of classes from the analyzed application. The length of these tuples is defined as 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 $\mathcal{I}$.

We use a simple example to clarify the terms and the definitions. Figure 6.1 introduces an example consisting of seven classes.

The key information of interest to us is the relationships that hold among classes. We have two kinds of class relationships:

- **Binary** relations R_B. These relations can be represented by a labeled pair $(C_i, C_j)_{Label}$, where C_i and $C_j \in C$, and C is the set of all the classes of the application. Considering the class diagram in Figure 6.1, *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*: $(A)_{Abs}, (X)_{Abs} \in R_U$.

Incidence Table 6.1 shows all the existing relations of order $o = 3$ in the example of Figure 6.1. Let's see briefly how the elements and properties are built.

Elements: Permutation of Classes.

We have said that the elements are tuples of classes. We build the elements as all the permutations of the classes with the length of order o:

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

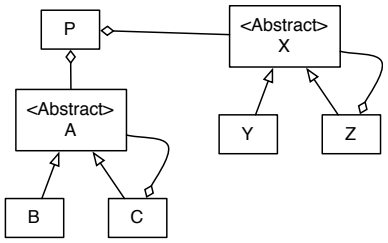

Figure 6.1: Example class diagram

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

Table 6.1: Order 3 context for the example in Figure 6.1

We use our variation of the algorithm proposed by Tonella and Antoniol [TA99]. It is an inductive context construction algorithm that avoids the combinatorial explosion which results when generating all possible tuples of classes. The underlying hypothesis is that the patterns consist of classes which are *all connected* together by their relations (unrelated classes are not interesting). 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.

Our variation in the algorithm is that, in all the steps, when we generate new class sequences we eliminate those class sequences that were already generated but with the classes names in different positions. For example, if the tuple {C A P} was generated, and we generate {A P C} or {C P A}, we only keep one of these as being representative of the three alternatives. We reduce the amount of tuples to minimize the amount of elements to be analyzed with the ConAn engine and also to reduce the redundancy of information between the elements. If we identify properties for the tuples {C A P}, the same properties are valid for the tuples {A P C} and {C P A} with minimal variations in the indices (See how the properties are built).

In case of Figure 6.1 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 elimination of repeated combinations of classes, the presence of disconnected subgraphs in the class diagram, and the partial connectivity inside each connected graph. Avoiding the combinations of a tuple makes the algorithm faster because we have even fewer elements as input.

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:

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

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

Each property has one or two indices that refer to the position of the class to be analyzed inside the tuple, and a subindex t to indicate the name of the property. For example, the property $(3,2)_{Sub}$ applied to the element {C A B} means that the class B is a subclass of the class A. Using indices instead of names allows disjunct tuples to share common properties. In the example result (Table 6.2) the tuples {C A B} and {Z X Y} have the common properties $(1,2)_{Sub}$, $(2)_{Abs}$, $(1,2)_{Acc}$ and $(3,2)_{Sub}$.

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

Table 6.2: Concepts of the example in Figure 6.1

6.2.2 ConAn Engine: 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 [SR97]. We use the Ganter algorithm [GW99], because it is one of the most efficient in terms of time performance [KO01].

The example of Figure 6.1 yields ten concepts for the order $o = 3$. They are listed in Table 6.2.

To make the explanation easier, let's see the specific case of a known pattern, such as *Composite Pattern* [GHJV95]. We can reduce and generalize the structural information to the relationships *isSubclass*, *isAbstract* and *accesses* and see it as in Figure 6.2. This simplified Composite Pattern is detected twice in the example of Figure 6.1: {C A B} and {Z X Y} as concept 2.

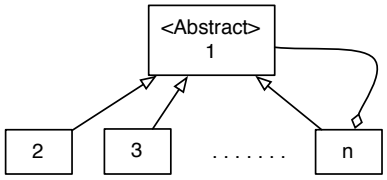

Figure 6.2: Structural relationships of the Composite Pattern

6.2.3 Concept Lattice: Post Filtering

Once the concepts are calculated, each concept is a *candidate* for a pattern. But not all concepts are relevant. Therefore a post processing of the concepts collection is needed to filter out concepts that are not meaningful. Let's look at the different filters.

Graph Representation of the Concept Intent.

To clarify the definitions, we introduce the graph representation of the intent of a concept. This graph is specific to our domain and cannot be applied to general FCA.

Intent relation graph: An *intent relation graph* is a graph whose nodes are the indices of the properties of the binary relation R_B and whose edges are binary relations R_B between the indices.

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

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

Considering Table 6.2, the binary relations of concept 2 are $(1,2)_{Sub}$, $(1,2)_{Acc}$ and $(3,2)_{Sub}$, of concept 4 are $(3,1)_{Sub}$ and $(2,1)_{Acc}$; and of the concept 8 is $(1,2)_{Sub}$. Thus, the intent graphs of concepts 2, 4 and 8 are shown in Figure 6.3. Considering the first diagram $c2$, the edge between node 1 and node 2 represents the property $(1,2)_{Sub}$ or $(1,2)_{Acc}$ and the second edge between node 2 and 3 is from the property $(3,2)_{Sub}$. Similarly, we build the graph of concepts 4 and 8. 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 are not related to the lattice edges.

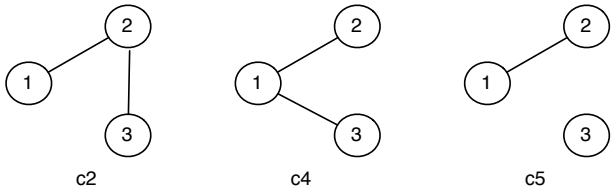

Figure 6.3: The intent graph of concepts 2, 4 and 8 of Table 6.2

Removing Disconnected Patterns.

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

Connected Pattern: A *connected pattern* is a pattern whose *intent relation graph* is connected.

Disconnected patterns are determined by a lower order context. A context of the order o is computed, when order o patterns are looked for, while $o-1$, $o-2, \cdots$ order contexts suffice for lower order patterns. In the example result (Table 6.2) the following concepts are disconnected: 6, 7, 8 and the top concept.

Merging Equivalent Patterns.

Suppose we have a system with the classes as shown in Figure 6.4. It might then happen that during the concepts and lattice calculation, we find the two concepts shown in Table 6.3.

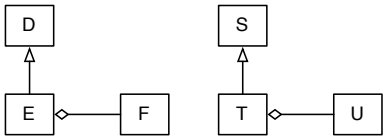

Figure 6.4: Adapter Pattern with two sets of classes

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

Table 6.3: Concepts of the example in Figure 6.4

Even though {D E F} and {T U S} are exactly the same pattern, the algorithm treats them separately. This happens because when generating the class sequences, we just keep one representative of each possible combination of classes. This means we just look at {D E F} which represents the class sequence {{D F E}, {E D F}, {E F D}, {F D E}, {F E D}}.

Equivalent Patterns: Two concepts, representing collaboration patterns, are *equivalent* if a permutation of the indices 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 [TA99].

In our example we find a permutation $\alpha = \{3 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3\}$, which transforms the tuple: {T U S} $\overset{\alpha}{\longmapsto}$ {S T U}. Concept c_2 (from Table 6.3) can now be removed when the translated extent of this concept is added on concept c_1. We call this process as *merging equivalent patterns*.
In the main example (Table 6.2) concepts 4 and 5 are equivalent: The permutation $\alpha = \{3 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2\}$ translates the properties of concept 5 into those of concept 4.

Applying these two filters (*removing disconnected patterns* and *merging equivalent patterns*) on the main example leads to the four patterns presented in Table 6.4. The first three patterns are directly taken from the first three concepts. Pattern p_4 is merged from concepts 4 and 5. The elements of concept 5 are translated into {P C A}, {P Z X} and {P Y X}, and are appended 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 6.4: Resulting Patterns after the merging of equivalent patterns from the concepts of Table 6.2

Guessing Names for Patterns.

Some of the detected patterns might structurally match known Design Patterns. To check this, we need a library of *named reference patterns*. In our approach, we provide this library by programming concrete instances of patterns. For the Composite Pattern an instance with the properties $(1,2)_{Sub}$, $(2)_{Abs}$, $(3,2)_{Acc}$ and $(3,2)_{Sub}$ for the order $o = 3$ is stored as a reference pattern (compare Figure 6.2). Thus pattern p_2 from our example is recognized as an instance of the Composite Pattern.

6.2.4 Pattern Neighborhood

One of the advantages of the FCA approach is that the generated concepts are related within a *complete partial order*. Thus given a concept *c*, we can identify the *cover concept* and *subordinate concept* of *c*. A *cover concept* in the lattice has less properties than *c*; and a *subordinate concept* in the lattice has more properties. Looking at the lattice, if a concept *c* has a cover concept *sp*, *sp* is higher up than *c*. Similarly, if a concept *c* has a subordinate concept *sb*, *sb* is lower than *c* in the lattice. With these two definitions, we define the idea of *neighborhood*. We define two kinds of neighbours. Considering that each concept *c* is a potential pattern, we define:

Almost pattern: An *almost pattern X* of a pattern *Y* is a pattern *X* which is contained in the cover concept of the pattern *Y* in the lattice.

Overloaded pattern: An *overloaded pattern X* of a pattern *Y* is a pattern *X* which is contained in the subordinate of a pattern *Y* in the lattice.

As a generic example, Figure 6.5 shows the structure of *almost* and *overloaded* patterns of a concept representing the structure of a *Composite Pattern*.

In the lattice of the example (shown in Figure 6.6), we see that concept 5 is an almost pattern of concept 1. For example, tuple {Y X P} is missing the property $(1,2)_{Acc}$, so belongs to concept 5 instead of concept 1.

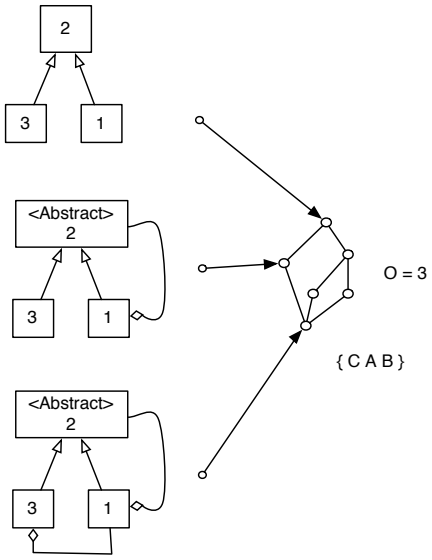

Figure 6.5: Almost and overloaded patterns of a Composite Pattern

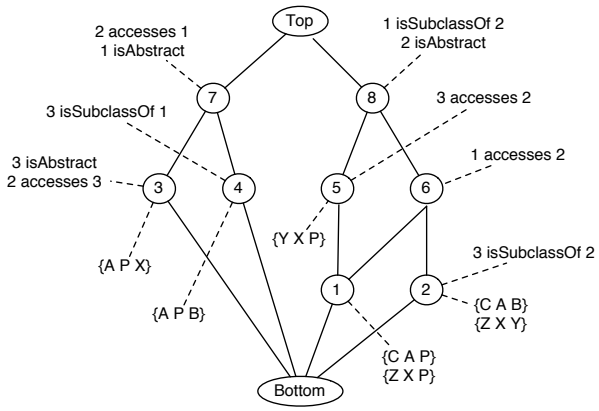

Figure 6.6: Resulting lattice of Incidence Table 6.1

But 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. This new concept (from the union of concepts 4 and 5) has two main consequences:

- the union of concepts 4 and 5 is now an almost pattern of concept 1
- new connections with other concepts can appear. In this specific case, if we translate the properties of concept 2 with the permutation $\alpha = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$, we see that this transforms the *union* of the concepts 4 and 5 in an almost pattern of the *new* concept 2 considering the intents of the concepts. But the union of the concepts 4 and 5 have to add the *transformed* elements of the extent of concept 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 $\{\text{Z X Y}\} \overset{\alpha}{\longmapsto} \{\text{X Z Y}\}$.

After the post-filtering process, where we modify the extent of some concepts, finding *equivalent patterns* and removing *disconnected patterns*, we no longer have a valid lattice, but simply a *partial order* [DP02]. Now the patterns have reached their final state and are listed in Table 6.5.

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 6.5: Final Patterns after applying the post filters to the concepts from Table 6.2

Linking Different Orders.

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

Cover pattern: C_2 is a *cover pattern* of the pattern c_1, if the *intent relation graph* 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 $o+1$.

Subpattern: C_2 is a *subpattern* of the pattern c_1, if the *intent relation graph* of the pattern c_2 is a subgraph of the intent relation graph of c_1.

Subpatterns 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 processing filter.

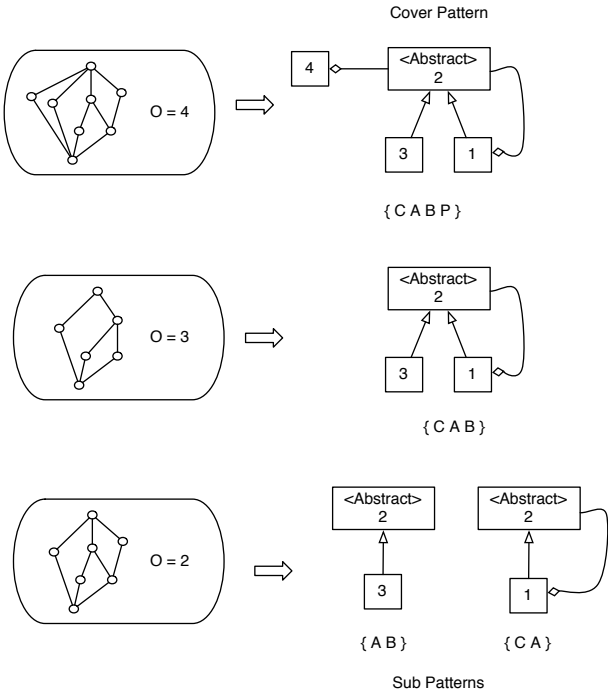

Figure 6.7: Sub and cover patterns of the Composite Pattern (p_2)

Figure 6.7 show the subpatterns of the Composite Pattern p_2 in the lower order $o=2$, and the cover pattern in order $o=4$.

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

Pattern Neighborhood: A *pattern neighborhood* of a pattern c is the union of the almost, overloaded, cover and subpatterns of the pattern c.

The approach of Antoniol and Tonella [TA99] deals with patterns as isolated entities. With our approach we relate the patterns within the same lattice and also with other patterns calculated in higher and lower orders. Thus, we are able to analyze not only the detected patterns but also the relationships to other patterns in the applications.

6.3 Validation: Case Studies

We have validated our approach by applying the tool we have implemented called *ConAn PaDi* to three mid-sized Smalltalk applications: ADvance, SmallWiki and CodeCrawler.

- **ADvance**[2] is a system round-trip engineering tool from IC&C. 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* to show the structure of the found patterns visually.
- **SmallWiki**[3] is a new and fully object-oriented wiki implementation in Smalltalk. It is highly customizable and easily extensible with new components, appearances, servers, storage, etc.
- **CodeCrawler** is a language independent software visualization tool[4]. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [Lan03].

Table 6.6 gives an overview of the size and processing time for each case study. We have applied the approach as outline in section 6.2 to each case study. After the concepts have been generated, we need a way to classify the patterns to analyze the applications. Thus we classify the patterns in terms of the properties that are used to describe them. We have built two *Classifiers A* and *B*. The *Classifier A* contains three properties: `isSubclass`, `hasAsAttribute`, `isAbstract` and the *Classifier B* contains two properties: `isSubclass`, `hasAsAttribute` (Table 6.7).

Thus, for example, we take the *Classifier B* and we get all the patterns that can be described with a set of properties that include `isSubclass` or `hasAsAttribute`. In the specific case of CodeCrawler in order= 3, we get 14 patterns which are distributed in 300 tuples of classes in total (Table 6.8). This means that *Classifier B* gives us an average of 21 tuples per pattern (300 / 14).

	ADvance	SmallWiki	CodeCrawler
Classes	167	100	81
Methods	2719	1072	1077
Lines of code	14466	4347	4868
Computation time	∼2 days	∼2 hours	∼30 min

Table 6.6: Statistical overview of the cases

Classifier A	Classifier B
isSubclass hasAsAttribute isAbstract	isSubclass hasAsAttribute

Table 6.7: Used Classifiers

The complete analysis of the quantitative impact of the classifiers is shown in the Table 6.8. A classifier with less properties gives a clearer image of the situation. Applying *Classifier B* with less properties heavily reduces the number of different patterns whereas the found patterns in total are much less reduced.

[2] http://www.io.com/∼icc/
[3] http://c2.com/cgi/wiki?SmallWiki
[4] http://www.iam.unibe.ch/∼scg

This leads to patterns which have more tuples as elements. The patterns of *Classifier A* are too "noisy". Comparing again the case of CodeCrawler in order=3, we get that the *Classifier A* has an average of 13 tuples per pattern (431 / 32) meanwhile the *Classifier B* has an average of 21 tuples per pattern (300 / 14).

		ADvance		SmallWiki		CodeCrawler	
		Classifier		Classifier		Classifier	
o		A	B	A	B	A	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 6.8: Classifier statistics

With 34 core classes of CodeCrawler we analyzed patterns of higher order $o > 4$. The statistics are in Table 6.9. Unfortunately this calculation takes one week on a PC.

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

Table 6.9: 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 6.10. *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 Gamma et. al.[GHJV95]. It is important to see that they are simplified to the used structural definitions 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 6.11 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 Classifier 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 Classifier A no *Subclass Chain* nor *Attribute Chain* of order $o = 4$ is found. Nevertheless Classifier B detects 12 instances of *Subclass Chain* and 15 instances of *Attribute Chain*. Classifier A does not detect those patterns because CodeCrawler has no chain with an abstract class *exactly* at the beginning and the rest of the chain consisting 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 cannot be detected because there is too much noise.

6.4 Application of the Approach: Analysis

In Chapter 3, we have discussed the approach in general terms. We have also mentioned different issues that the user must take into account when applying the approach. Here following we list the issues concerning *Collaboration Patterns* on complete applications.

- *Choice of Elements and Properties.* Elements are tuples of classes built from the metamodel. As

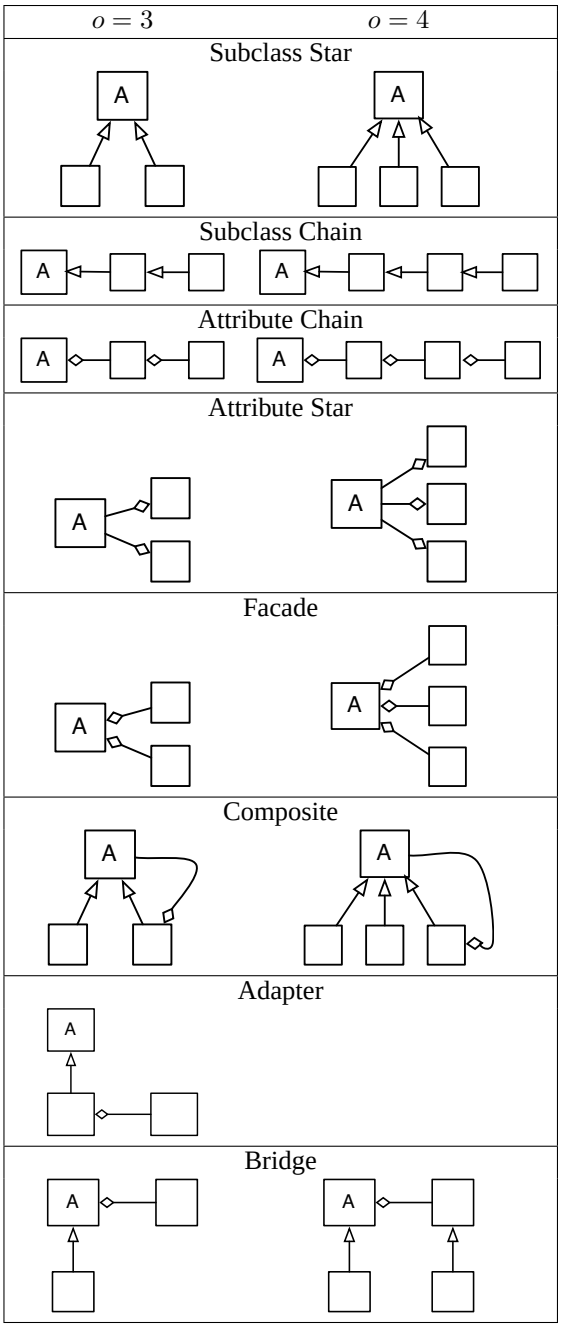

Table 6.10: Structure of investigated patterns

our case study refines the work of Tonella and Antoniol [TA99] we use the same idea to build the elements. The choice of properties is the set of the structural relationships to characterize *Structural Design Patterns* [GHJV95]. Except the properties *is subclass of* and *is abstract* that are mapped directly from the metamodel, the rest of properties are computed from the metamodel.

- *Compact Representation of Data.* This issue is related to how the tuples of classes are generated. We avoid generating all permutations of class sequences in the tuples. For example, if the tuple {C A P} is generated, and we subsequently generate {A P C} or {C P A}, we only keep one of these as being representative of all three alternatives.

		ADvance		SmallWiki		CodeCrawler	
		Classifier		Classifier		Classifier	
o	Pattern	A	B	A	B	A	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 6.11: Investigated Patterns

- *Multiple Contexts.* In this case, we have used 3 lattices. Each lattice is for one order of the elements: 2, 3 and 4. All of them use the same set of properties.
- *Performance of the Algorithm.* With the tuples of classes of order more than 4, we are not able to have a reasonable computation time of the algorithm. With one of the applications using tuples of classes of order 4, the computation time took around 2 days and this is not acceptable time from our viewpoint to software engineering.
- *Meaningless Concepts.* As we said previously, each concept is a candidate for a pattern. In this case, each concept represents a graph in which the set of the elements is the set of nodes, and the set of properties define the edges that should connect the nodes, *e.g.*, if (A P C) has the properties *A is subclass of P* and *P uses C*, then we have a graph of 3 nodes with edges from A to P and from P to C. Thus we discarded all the concepts that represent graphs in which one or several nodes are not connected at all with any node in the graph.
- *Mapping Partial Order of the Lattice.* In this case, we map the partial order of the lattice to the definition of *neighbors* of a pattern. We can detect either missing relationships between classes needed to complete a pattern, or excess relationships between classes that extend a pattern.
- *Limits of* Collaboration Patterns. In this high level view, we consider that there are still possible new collaboration patterns to detect when applying the approach in other applications.

6.5 Discussion

Based on the results, we are able to evaluate our approach from different viewpoints:

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.

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.

Identify the possible presence of classical design patterns: Candidates for classical design patterns are found. Some of these turn out to be *false candidates*, *i.e.*, structural patterns which superficially resemble design patterns, but are not in fact instances of those design patterns. The reasons for the misinterpretation are:

- Not all the properties are absolutely reliable. For example, the extraction of the type of an attribute is based on a heuristic [Aeb03], because we work with Smalltalk, which is a dynamically typed language
- The collaborations of the detected structural patterns match that of the known design pattern, but not its intent. This happens mainly with the *Facade*, *Adapter* and *Bridge* patterns in our case studies. Consider, for instance, the *Bridge* pattern of order $o = 3$ in Table 6.10. A class that has a subclass and accesses another class is a *candidate* for a *Bridge*, but there is no guarantee that such a class is actually serving the purpose of a *Bridge*.

To decide which are real classical design patterns, detailed knowledge about the application is needed. It cannot be decided with the information provided by *ConAn PaDi*. We were just able to detect structural Design Patterns 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 [TA99].

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.

Find candidate classes for restructuring: In our case studies we could not find a candidate for restructuring, because our analysis was made using only structural information. In general, control flow information and detailed information about the method bodies is needed to detect candidates for restructuring [Tic01]. 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.

Mining patterns: As our approach with FCA detects *any* kind of patterns we found plenty of "new" patterns, meaning that they are not referred to in the literature. Whether those patterns are useful and make sense as Design Patterns is another issue. If we see in Table 6.8 the case study ADvance for the order $o = 4$, we detect 329 different patterns. The frequency of several patterns is shown in Table 6.11. We see that the *Facade* is detected 627 times and the *Bridge* only 20 times. The accumulated elements of the pattern are shown again in Table 6.8. 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: if we 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: $\{C_0C_1C_2\}, \{C_0C_1C_3\}, \{C_0C_1C_4\}, \cdots, \{C_0C_4C_5\}$.

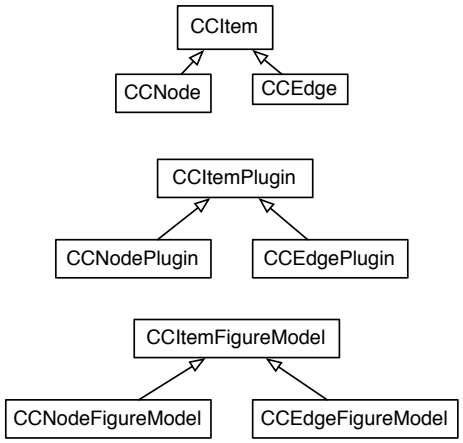

Figure 6.8: Three *Subclass Stars* of CodeCrawler

Understand the class roles: Using the information gained 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 6.8. The assumption that these are three parallel structures cooperating together are proved by future investigation.

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.

6.6 Related Work

The starting point for our work was the approach of Tonella and Antoniol [TA99], and we have summarized our improvements to their approach in Section 6.2. Other related work focuses on the detection of design patterns *à la Gamma* as opposed to more general *software* patterns. We cite some of these approaches.

Brown [Bro96] presents in his Masters 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 Corman's cycle-detection technique [CLR90]. There is no general abstraction proposed to encode patterns. In particular, Brown does not demonstrate a clearly generalizable approach to detect patterns: for each pattern, a specialized detection algorithm must be developed.

Seemann et. al. [SvG98] use a compiler to generate graphs from the source code. This graph acts as the initial graph of a graph grammar that describes the design recovery process. The validation is made with respect to well-known design patterns such as *Composite* and *Strategy* in the Java AWT package.

Keller et al. [KSRP99] present an environment for the reverse engineering of design components based on the structural descriptions of design patterns. Their 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.

Niere et al. [NSW+02] 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. An evaluation of the approach is made with the Java AWT and JGL libraries.

In Krämer and Prechtel's approach[KP96], the patterns are stored as Prolog rules. Their Pat tool takes the meta-informa–tion directly from the C++ header files and queries them. The validation on the C++ libraries

shows that the precision is around 40 percent.

Albin-Amiot et al. [AACGJ01] show how to automate the instantiation and detection of design patterns. To cope with these objectives, they define a *Pattern Description Language* to describe design patterns as first-class entities. Thus they can manipulate and adapt design patterns models to generate source code. With the same language, and a Constraint Satisfaction Problem formalism to detect complete and distorted versions of design patterns, and with the tool *JavaXL* (their own tool) they are able to make the distorted versions compliant with the design patterns models. With this approach, they help OO software practitioners design, understand, and re-engineer a piece of software, using design patterns.

6.7 Conclusions

This section summarizes this approach with a summary, a discussion, the answers to the research questions proposed in Chapter 2 and future work.

6.7.1 Summary

The complete description of the approach including the analysis of the cases studies and the tool implemented to support it is described in [Buc03]. Although our work is based on that of Tonella and Antoniol, there are some notable differences:

- According to our measurements of Tonella and Antoniol's algorithm, the performance with our data was a critical issue. We propose an improvement to their algorithm to make it faster. We eliminate the redundancy in the sets of elements considered to reduce the calculation time for the formal context generation. Using as example Figure 6.1 (Section 6.2.1) where there are 7 classes and running on the same platform, we have made a comparison (shown in Table 6.12) in terms of time performance.

order	our approach		[TA99]	
	Number of Tuples	time [s]	Number of Tuples	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 6.12: Comparison between our inductive approach and the inductive approach from Tonella

  In Table 6.12, we see that if we compare our approach to that of Tonella and Antoniol, the calculation time in the different orders is uniform, whereas in their approach it increases for each order. The number of tuples also increase with each order in their approach, whereas it remains relatively constant in ours.

- With our improvement in the algorithm where we keep one representative of the set of possible combinations of a class sequence, we avoid repeated information and we avoid removing *equivalent instances* [TA99] inside the concepts.
- With our approach, we are not constrained to the detection of *design* patterns. We are focused on the larger scope of detecting recurring collaborations patterns implicit in the code, which we refer to as *collaboration patterns*. These collaborations may represent design patterns, architectural constraints, or simply idioms and conventions adopted for the system.

- In contrast to Tonella and Antoniol's approach, we relate the patterns to each other using the connections between the concepts given by the partial order in the lattice, and the lattices calculated with the different *orders*. This is what we named *Pattern Neigborhoods*. For example, it is possible to detect patterns which are *almost* like another pattern.
- We propose to take the information from a language independent Meta-Model instead from the source code itself. This makes the approach more general because it can be applied to applications in different programming languages. In contrast, Tonella and Antoniol focus on experiments done with C++ applications.
- We separate the calculation of the patterns from the analysis process. This allows us to separate the time-consuming calculation of the patterns from their analysis. The patterns are stored in a repository, and can be analyzed by applying different classifiers to the results.
- To gain an overview more quickly, as a starting point, we compare the detected patterns against a reference library of well-known design patterns in the post-processing phase. This library is incremented with new detected patterns with each application we analyze.

6.7.2 Research Questions

Based on the approach, we are able to answer some of the research questions proposed in Chapter 2:

- *How does the technique help in discovering patterns introduced in the system ?*

  In this last approach, we confirm again that the application of FCA is scalable because in this third abstraction level we analyze a complete application. Using classes as FCA elements and structural relationships as FCA properties, we have identified recurring situations (named as *Collaboration Patterns*) in a complete application. These *Collaboration Patterns* reveal existing design patterns, coding styles or architectural patterns implemented in the system. With this information, we have a complete mental model (from the structural viewpoint) of a system, complemented with additional issues such as classes that are used in all the system, presence of design patterns and classes roles.
- *Does the technique discover known and unknown patterns ?*

  Both kinds of patterns are discovered. The detected design patterns are considered as known patterns and the architectural patterns and coding styles are considered as unknown patterns in our work context. The last ones are considered as unknown because in the most of the cases they are not documented, and the name of methods of involved classes do not provide useful information. In the case of design patterns, in spite that it can be implemented without following naming convention, they can be characterized by a specific set of properties.
- *How do the patterns help to understand a system ?*

  This question is related to the first one. As we have said previously, with the identified patterns we get a complete mental model (from the structural viewpoint) of a system complemented with additional issues such as related patterns, classes that do not participate in any pattern and used coding styles.

6.7.3 Future Work

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

Solve the scalability problem. In an industrial environment *ConAn PaDi* would be considered to be too slow. Results should be available in real time or at least within seconds, otherwise the developer will not use this tool. One idea to improve the speed is not to take all orders into consideration. Figure 6.9 shows approximately the expected found patterns in correlation with the order. We suppose that we just calculate

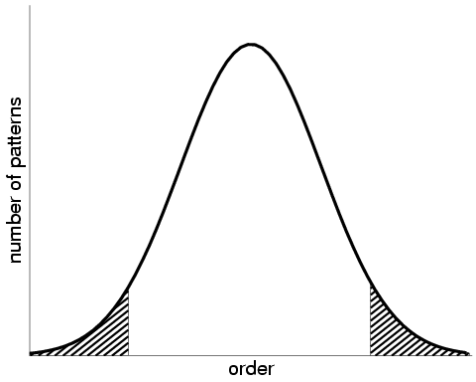

Figure 6.9: Unproblematic orders for calculation

the patterns before and after the peak of the curve. We expect that the patterns for the lower orders are interesting to analyze the dependencies and those for bigger orders are interesting to understand the roles and gain an overview of an application.

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

Chapter 7

Conclusions

The last chapter summarizes the main features, the answers to the research questions proposed in Chapter 1, the main lessons learned, and future work related to the approaches developed in this thesis.

7.1 Summary

In this thesis we present a general approach for applying FCA in reverse engineering of object-oriented software. We also evaluate the advantages and drawbacks of using FCA as a metatool for our reverse engineering approaches.

We also identify the different bottlenecks of the approach. Thus, we are able to focus clearly on solving which and where the limitations appear (if there are some possible solutions) to draw the maximum benefit offered by FCA.

From our tool builder viewpoint, we have proven that FCA is an useful technique to identify *groups* of software entities with hidden dependencies in a system. With FCA, we have built different software engineering tools that help us to generate *high level views* at different levels of abstraction of a system. We generate the *high level views* because without them, the *software engineer* should be obliged to read the lattice. This can represent a problem because in most of the cases, besides the useful information, the lattice can also have useless information that can introduce *noise* in analyzing a system.

7.2 Research Questions

Based on the approach, we are able to answer some of the research questions proposed in Chapter 1:

- *Do we understand how the system is implemented? What are the constraints or limits imposed in the system?*

  We have shown in this thesis that the different *high-level views* (*XRay views* at class-level, *Hierarchy Schemas* at class hierarchy-level and *Collaboration Patterns* at system-level) make *hidden* dependencies as *explicit* ones. These dependencies reveal the different existing contracts between the different (groups of) software artifacts in a system. Depending on the abstraction level of the system we analyze, with the high-level views we identify different characteristics in a system. Just to mention some of them, for example, we identify group of collaborating methods at class-level, misuse of inheritance mechanisms at class-hierarchy or use of patterns at the application level. We believe that each view provides enough (but not yet complete) information to generate the first fingerprints of a system.

- *How can we detect implicit unanticipated dependencies?*

  When defining adequate properties in a FCA context (in the iterative *modelling* process of elements and properties shown in Chapter 3), the combination of them (intents of concepts) are the potential sources of information to detect implicit dependencies. Once we have applied the *Post-Filtering* process (shown in Chapter 3), the resulting concepts are those that show meaningful implicit unanticipated dependencies.

- *How are the mechanisms such as polymorphism and inheritance used in the system?*

  In *Hierarchy Schemas* and *Collaboration Patterns*, we have analyzed inheritance as a building mechanisms in a system. In Hierarchy Schemas we have shown we can identify *good*, *bad* and *irregular* situations of the use of inheritance relationship. In spite that this relationship defines a contract between several classes in terms of a *is-a* relationship, we have seen that the misuse of this mechanism can produce *noise* in a class hierarchy. The *bad* and *irregular* cases reveal us for example, violation of class encapsulation or bad reuse of behavior of parent classes. In the case of *Collaboration Patterns*, the inheritance mechanism was used in combination of other properties to identify patterns. Half of the identified patterns included the inheritance relationship, meaning that it represents an often-used mechanism in a system.

  In the case of polymorphism, this characteristic was not a main aspect in our analysis, meaning that we did not use it as a defined property in our contexts. The use of polymorphism is part of the analysis of systems in future work.

- *How can we discover defects introduced in the systems?*

  Discovering defects in systems in our approach depends also in identifying adequate properties. The combination of properties in all the approaches helps us to reveal good as well as bad situations in a system. This case is proven in the case of hierarchy schemas where we identify classes that were using the state of superclasses without using the accessors, and consequently violating the superclass encapsulation.

- *How can we discover recurring situations ("patterns-like") of dependencies in the systems?*

  When using FCA, a concept reveals us a set of elements that share a set of properties. The most meaningful concepts are those that have a significant number of elements. These concepts then show that we have a number of software artifacts that share some commonalities, that we consider them as patterns. They represent recurring situations appearing in a system.

- *Is the mental model generated by "high-level views " meaningful enough (in terms of information) to understand the system?*

  As we have said in the first answer, we believe that each view provides enough (but not yet complete) information to generate the first fingerprints of a system. We think that we still need more case studies in each approach to identify which are the missing characteristics in a system to get a more complete first mental model of a system.

- *Which are the advantages and disadvantages of using a clustering technique such FCA as a metatool?*

  In the next section of this chapter, we have summarized which are the advantages and disadvantages of using FCA as a metatool. Among the main issues, we consider that there are three ones worth mentioning:

  – The different bottlenecks produced by the computation times of the algorithms and building the FCA elements and properties can compromise the success or fail of an experiment.
  – The noise produced by the amount of concepts can bring useless information for our approaches.
  – The combination of properties help us detect unanticipated dependencies revealing *hidden* contracts between different software artifacts. This was the most valuable information in our approaches.

7.3 Lessons Learned

In general terms, we have seen that Formal Concept Analysis is a useful technique in reverse engineering. From our experiencies [Aré03, ABN04, ADN03] in developing this approach, several lessons learned are worthwhile mentioning.

Lack of a general methodology. The main problem we have found in the state of the art is the lack of a general methodology for applying FCA to the analysis of software. In most publications related to software analysis, the authors only mention the FCA mapping as a trivial task, and how they interpret the concepts. With our approach, we achieved not only to identify clear steps for applying FCA to a piece of software but where we have identified different bottlenecks in using the technique.

Modelling software entities as FCA components. The process of modelling software entities as FCA components is one of most difficult tasks and we consider it as one of the critical steps in the approach. *Modelling* is an iterative process in which we map software entities to FCA components, and we test whether they are useful enough to provide meaningful results. This task is not trivial at all, because it entails testing at least 5 small case studies (which should be representative of larger case studies) in each developed approach. Each case study should help to refine the building of FCA components to obtain meaningful concepts. Thus, this process can be considered as incremental step-by-step one. Obviously, we can conclude after the initial experiments that the chosen FCA components are not the most adequate and discard the complete FCA mapping, and start the process from the scratch.

Performance of FCA algorithms. Computation Time The performance of the algorithms (to build the concepts and lattice) was one of main bottlenecks. In small case studies, this factor can be ignored because the computation time is insignificant. But in large case studies this factor can cause the complete approach to fail because computing the concepts and the lattice may take several hours (eventually days) and this is not a acceptable time to do some software analysis.

Performance of FCA algorithms. Implementation The computation time of the FCA algorithms is also affected by how they are implemented in a chosen language. A direct mapping of the algorithms (in pseudo-code, as they are presented in books) to a concrete programming language is not advisable. In our specific case, we took advantage of efficient data structures in *Smalltalk* language to represent the data and improve the performance of the algorithms.

Supporting Software Engineers. The result of our experiences must be read by software engineers. One positive issue in this point is that with the high level views the software engineer is not obliged to read the concept lattices , meaning that he needs not be a FCA expert. Our approach is not focused on end-users. We would need a new abstraction level over the approach with a more friendly software interface.

Interpretation of the concepts. Although we can have an adequate choice of FCA elements and properties, the interpretation of the concepts is a difficult and consuming-time task to achieve. In most of the cases, we have tried to associate a meaning (a name) to each concept based on the conjunction of its properties. This task must be done by a software engineer applying *opportunistic code reading* to get meaningful interpretations. It is difficult to discover which combination of properties are or not useful. This process is completely subjective because it depends on the knowledge and experience of the software engineer.

Use of the Complete Lattice. Not all the concepts have a meaning in our approach, so we do not use the complete lattice in our analysis. In most of the cases, we remove meaningless concepts because from software engineering viewpoint they did not provide enough information for the analysis. We hypothesize that in certain cases it may be possible to use the complete lattice, but we did not find any.

Use of the Partial Order. Another critical factor is the interpretation of the partial order of the lattice in terms of software relationships. Only in *Collaboration Patterns* we were able to obtain a satisfactory interpretation of the partial order. Thus we can conclude that the interpretation is not a trivial task.

Metamodel Information. The results can only be good and complete as the information from the metamodel is. When the input for FCA is already not complete or reliable, the percentage of useless results can be higher.

7.4 Future Work

The future work is focused on several research directions, that consist mainly of:

- Analysis of a system using other abstraction levels to analyze how useful the developed approach is.
- Refinements and improvements in the approach to adapt it to new abstraction levels of analysis of a system.
- Tests with other concept and lattice building algorithms to compare (and eventually improve) the existing computation time of the ConAn engine.
- Development of a measurements framework to measure the different concept and lattice building algorithms based on the used data in our approach.
- Analysis of the partial order of resulting lattices to get possible mappings in terms of software engineering relationships.
- Adding more domain information to the existing approaches to evaluate if they can provide more information about the analyzed applications.

Appendix A

ConAn Framework

A.1 Introduction

To validate the application of Formal Concept Analysis in generating *High Level Views*, we have built three tools based on a framework named as ConAn. The name is an acronym for *Con*cept *An*alysis. In this appendix, we explain in detail which are the different components of the framework.

A.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*) [DGLD04][Tic01].

Moose is a reengineering research platform implemented in VisualWorks [DGLD04][Tic01]. 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 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

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

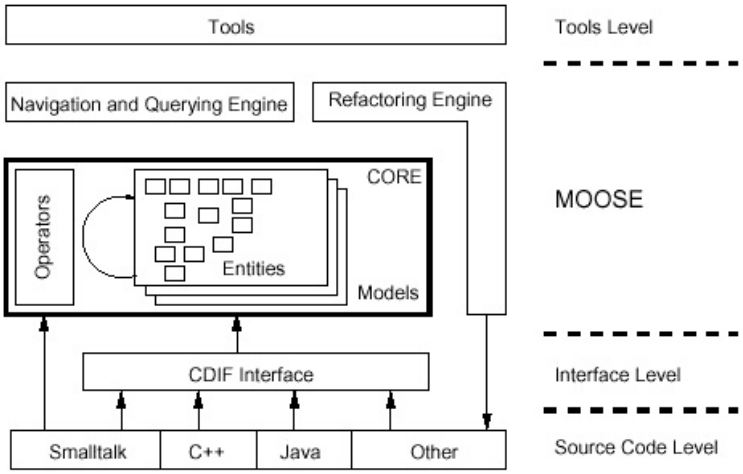

Figure A.1: *Moose* architecture.

A.3 ConAn: a Framework for FCA

ConAn is a general, extensible framework for FCA implemented in VisualWorks 7.0 [Vis03]. Its name is an acronym for *Con*cept *An*alysis. It allows the user to define any kind of elements and 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.

A.3.1 Features of ConAn

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.

A.3.2 Components of ConAn

The process from the input to the output provided by this framework is defined by 5 phases defined in the Figure A.2 (introduced in Chapter 3)

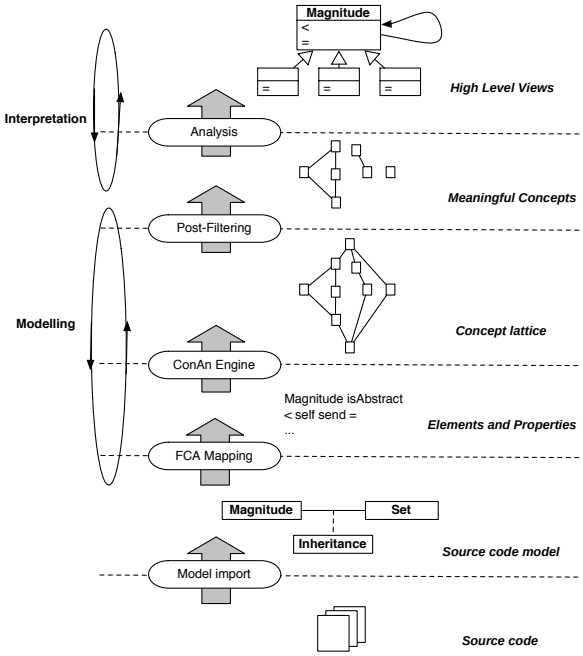

Figure A.2: The overall approach

We limit our explanation to the phase that are (semi)automatic processes. The phase *Model Import* are not part of our framework. It is provided by MOOSE framework (explained in Section A.2).

- *FCA Mapping*
  - *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.
- ConAn Engine
  - *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* [SR97] and *Ganter algorithm* [GW99].
  - *Calculation of Lattice*
    The lattice is built according to the complete partial order of the concepts. So far, the framework only provides one algorithm. 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 extent 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. 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] [Lan99].
- *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 [LRP95]. 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 [Fur86]. 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.

A.4 Fish Eye View

This section explains how the *Fish Eye View* is applied in *ConAn*. This part of the tool was used mainly in detecting *Collaboration Patterns*.

The *Fish Eye 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.

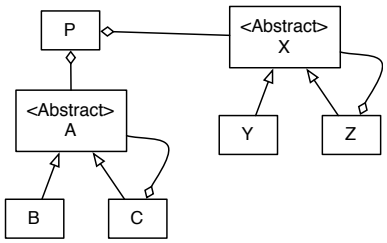

Figure A.3: Example class diagram

If we take as an example the concept 5 of the lattice shown in the Figure A.4, we will see the concepts as in Figure A.5

The two lists in the middle are the extent (elements) and intent (properties) of the concept five. The item in the top list is the edge to concept one. 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

[2] http://www.iam.unibe.ch/∼scg

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

Table A.1: Order 3 context for the example in Figure A.3

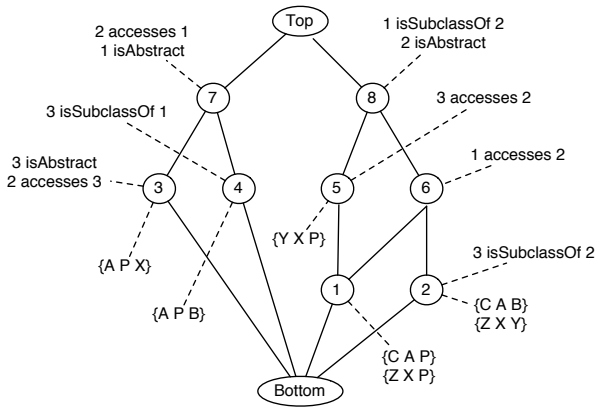

Figure A.4: Resulting lattice of Incidence Table A.1

in the bottom list is the edge to concept number eight. It removes 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 the respective clicked concept. Thus the user can navigate through the concept lattice.

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 A.6.

A.5 User Interface

ConAn provides three tools to analyze the *XRay Views*, *Hierarchy Schemas* and *Collaboration Patterns*. In each case, the tool is composed of two main parts: *Importer* (that allows the user load class(es) to analyze) and *Visualizer* (that allows the user analyze the results).

A.5.1 User Interface of XRay Views

Importer

The XRay views analyzes a class as a sole development unit. To analyze a class, we need to import it to the metamodel. Figure A.7 shows the importer of classes for XRay views. It is worth mentioning that the importer is adapted to load Smalltalk classes, but it can be adapted to read any object-oriented languages.

In (1) all VisualWorks packages are listed. The user selects the package in which the class is contained in. Once the package name is selected (in this specific case, Collections is selected), in (2) all the classes

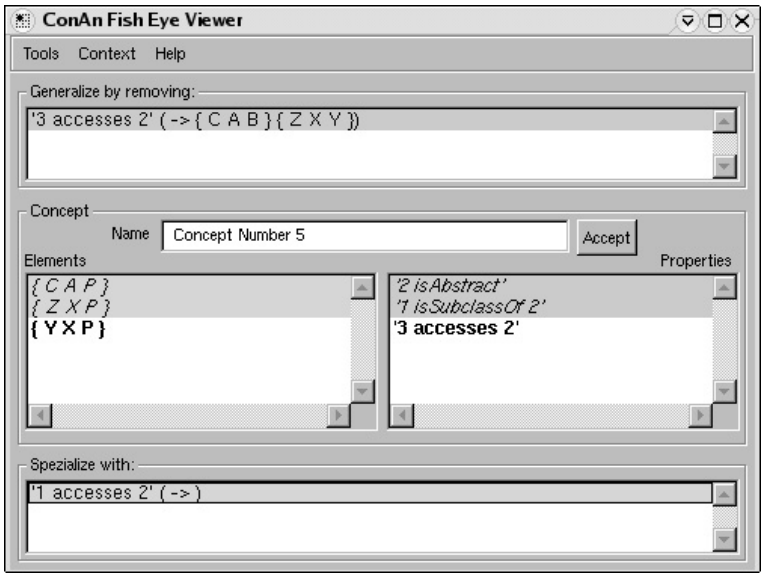

Figure A.5: Implementation of the *Fish Eye View* in ConAn

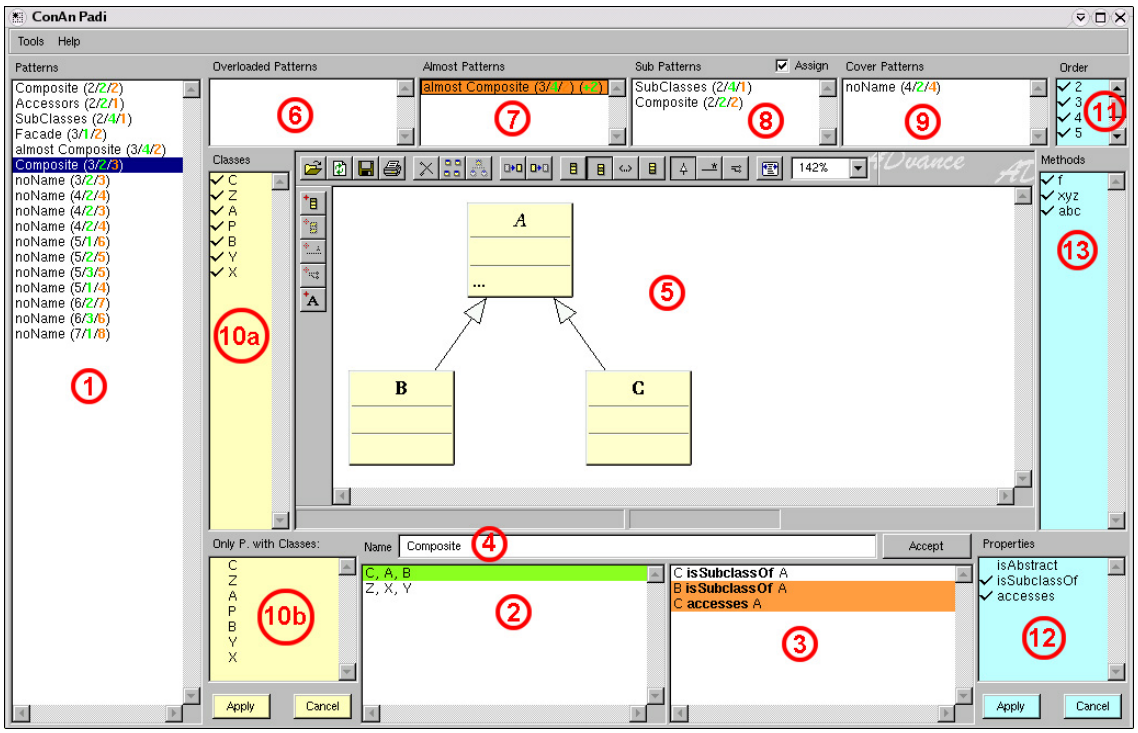

Figure A.6: *ConAn PaDi* with the result from the classes of Figure A.3

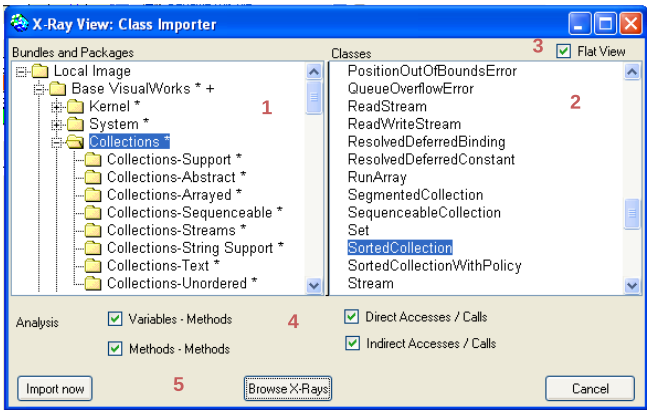

Figure A.7: Importer of classes in XRay Views

contained in the package are listed. In this window, the user can choose to see the classes as a list (Flat View is selected (3)) or to see the class ordered in the respective class hierarchy which the class is contained in (Flat View is not selected (3)). Once the class is selected (in this specific case SortedCollection), the user chooses in (4) if the analysis is focused on direct and/or indirect attributes' accesses and/or methods' calls. Then the button Import now is selected to import the class, to generate the concepts and the lattice. Then the button Browse X-Rays is selected to open the visualizer with the groups of XRay views. The button Cancel cancels all the described processes and closes the window.

Visualizer

The visualizer is composed of two main windows shown in the Figure A.8. In (1) all the identified concepts are listed. They are classified according to the functionality they represent in the class (*e.g.*, Collaborating Attributes or Stateful Core Methods). When the concept is selected, the *Elements* and the *Properties* are listed (in this specific case the elements are DefaultSortBlock and sortBlock and the properties are isUsedExternally, isUsedIn: SortedCollection → representBinaryOn: and isUsedIn: SortedCollection → initialize).

Then the graph of all the method calls and attributes accesses of the class is generated. Then the subgraph, where the elements and the parameters of the properties are contained in, is generated and shown in (2). The methods and attributes named in the selected concept are represented as colored nodes in the subgraph.

A.5.2 User Interface of Hierarchy Schemas

Importer

The *Hierarchy Schemas* analyze a complete hierarchy. Thus we need to import it to the metamodel. Figure A.9 shows the importer of class hierarchies for *Hierarchy Schemas*. It is worth mentioning that the importer is adapted to load Smalltalk classes, but it can be adapted to read any object-oriented languages.

In (1) all VisualWorks packages are listed. The user selects the package in which the class root of the class hierarchy is contained in. Once the package name is selected (in this specific case, Collections is selected), in (2) all the classes contained in the package are listed. In this window, the user can choose to see the classes as a list (Flat View is selected (3)) or to see the class root ordered in the respective class hierarchy which it is contained in (Flat View is not selected (3)). Once the class root is selected, the button Import now is selected to import all the classes of a hierarchy that has the selected class as a root. Then the concepts and the lattice are generated. Then the button Browse Hierarchy Schemas is selected to open the visualizer with the identified schemas. The button Cancel cancels all the described processes and closes the window.

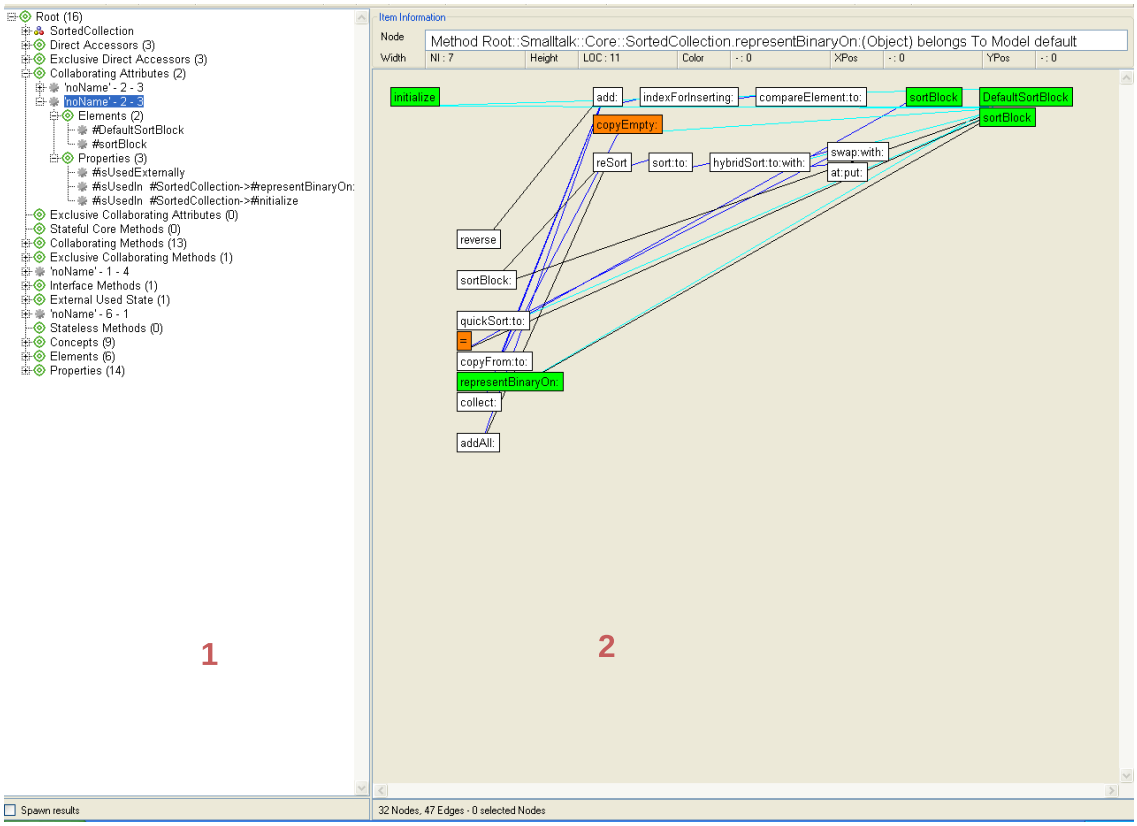

Figure A.8: Visualizer of XRay Views

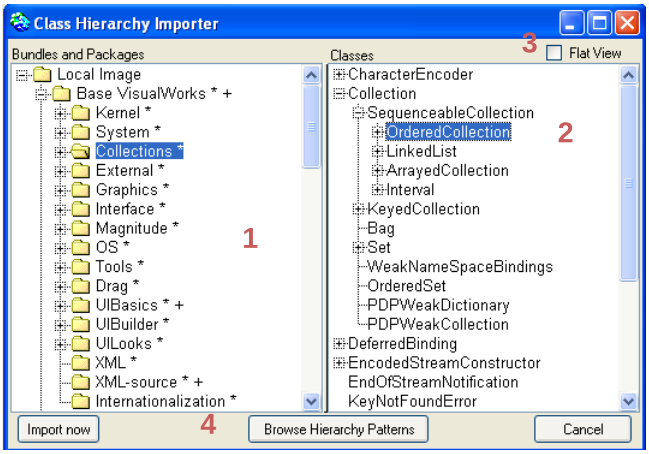

Figure A.9: Importer of Class Hierarchies in Hierarchy Schemas

Visualizer

The visualizer is composed of two main windows shown in the Figure A.10. In (1) all the identified schemas are listed. In each schema, all the classes where the schema appear are listed. Once the user selects a class (in this specific case SortedCollection), the elements and properties of the schema are listed (in this specific case, the elements are firstIndex and lastIndex and the properties are accessStateDefinedInAncestor: OrderedCollection, not-isDefinedInDescendant and not-isDefinedLocally).

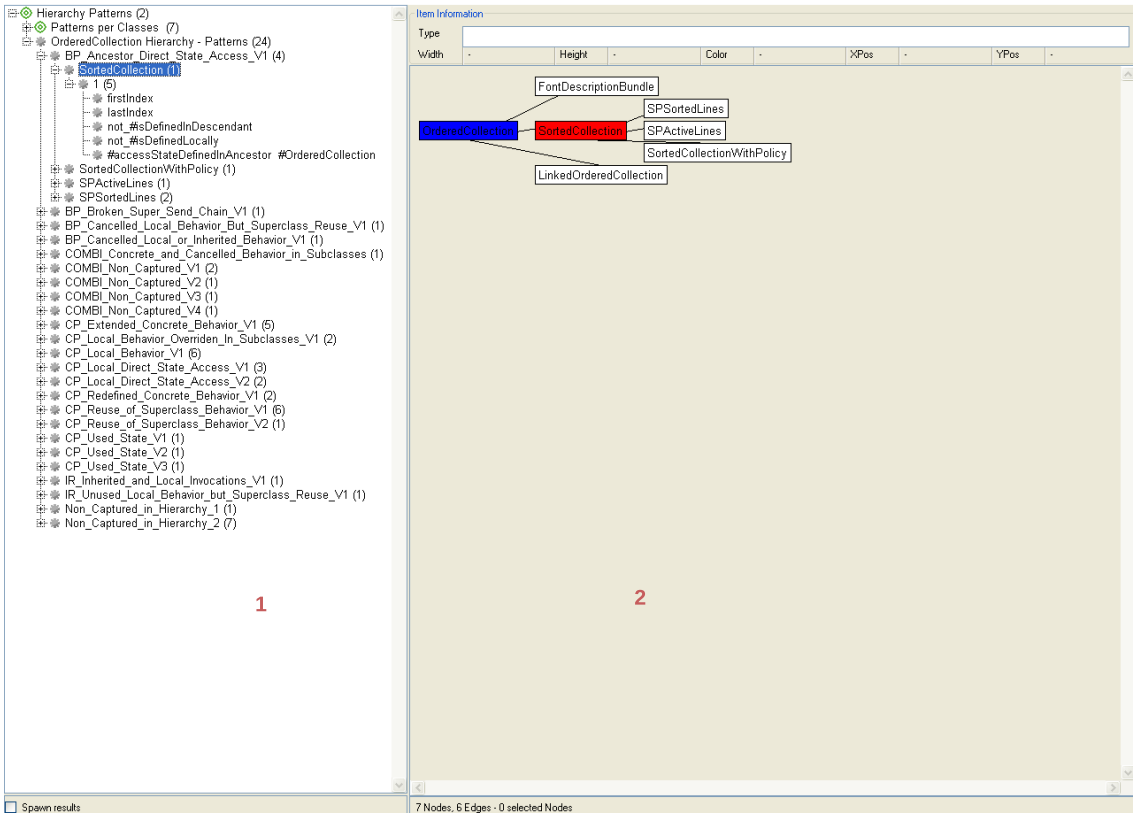

Figure A.10: Visualizer of Hierarchy Schemas

Once the class is selected and the elements and properties are listed, in (2) a graph with all the class hierarchy is generated. Then, all the classes contained in the selected concept (listed as elements and as parameters of the properties) are colored in the visualized class hierarchy.

A.5.3 User Interface of Collaboration Patterns

Importer

Before the patterns can be explored they have to be loaded in the repository. The *ConAn PaDi* Importer (Figure A.11) 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.

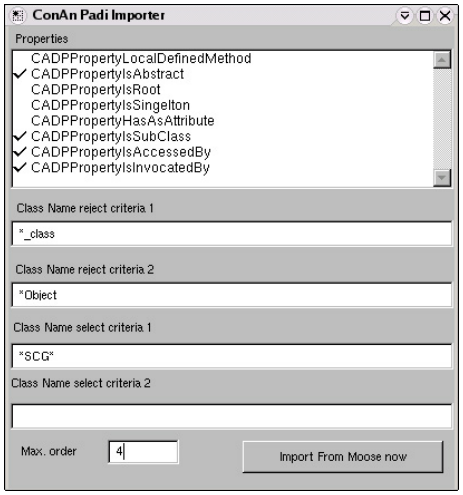

Figure A.11: Importer of classes of Collaboration Patterns

Visualizer

Once the classes have been imported, all the patterns are calculated, the user visualizes the results in the Pattern Browser shown in Figure A.6.

The list of all the patterns (1) shows their names followed by three numbers. The first number shows from which order o they are. The second number indicates the number of elements the pattern contains and the last 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 [OMG99] diagram using ADvance [IC 01][Buc02]. 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 A.6 *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).

Appendix B

Introduction to Formal Concept Analysis

B.1 Introduction

This appendix is an introduction to the main terminology of Formal Concept Analysis. It is attempted to give a global overview in this mathematical discipline to understand how FCA works, how it can be applied in different case studies and and to get the idea how the ConAn framework was developed. It is a summary of the definitions given in [SR97], [ST98] and [GW99].

Formal Concept Analysis (FCA) [Bir40], [BM70] [GW99](also known as Galois lattices [Wil81]) is a branch of lattice theory that allows us identify meaningful groupings of *elements* (referred to as *objects* in FCA literature) that have common *properties* (referred to as *attributes* in FCA literature) [1].

In all the extent of this report, we use one illustrative example about a crude classification of a group of mammals: *cats, gibbons, dolphins, humans,* and *whales*, and we consider five possible characteristics: *four-legged, hair-covered, intelligent, marine,* and *thumbed.* Table B.1 shows the relationships between the mammals and its characteristics.

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

Table B.1: Mammal example: Table $\mathcal{T}$ represents the binary relations

But first of all, we need to understand a few definitions to see how we analyze the information provided by this technique. The symbols $\cap$, $\cup$, $\setminus$ $\in$, $\subset$, $\subseteq$ (used in the rest of the chapter) represent the classical operations on sets: *intersection, union, complement, belongs to, restrictive inclusion, inclusion.* The rest of the symbols that have a specific meaning in this context are introduced in the text.

[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

B.2 Context and Concepts

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

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

$\mathcal{E}$ 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 $\mathcal{P}$: $\mathcal{I} \subseteq \mathcal{E} \times \mathcal{P}$ and is usually represented as a cross-table $\mathcal{T}$. The binary relation in our example is shown in the Table B.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 their composition is *extensive*:

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

In the mammal example:

$$\sigma(\{Cats, Gibbons\}) = \{hair\text{-}covered\} \text{ and } \tau(\{marine\}) = \{dolphins, whales\}$$

Based on the previous definitions, we define the term of 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]:

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

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. Table B.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 $\mathcal{T}$.

[2] The notation in [GMM+98] and [GW99] 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 $Y \subseteq \mathcal{P}$:

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

Thus using this notation, [GW99] summarizes the naive possibilities of generating all concepts as: *Each 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 the report the notation with σ and τ because it helps us to distinguish between the set of elements and properties.

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 B.2: Concepts of the mammal example

B.3 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 $(X_0, Y_0) \leq (X_1, Y_1)$, 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 concepts constitutes a *concept lattice* $\mathcal{L}(\mathcal{T})$ [Bir40]. The concept lattice for the mammal example is shown in Figure B.1

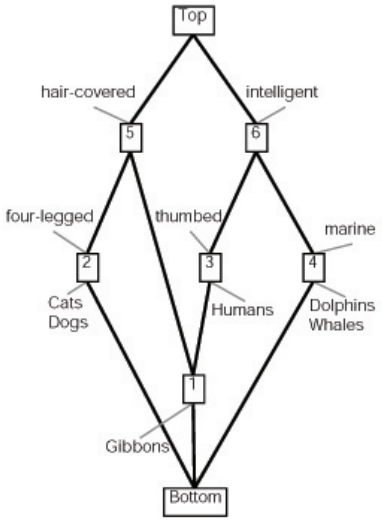

Figure B.1: The lattice of the mammals example with classical notation.

Each node in the lattice represents a concept and they are shown in Table B.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

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

Following our mammal example, let's see the results of $c_3 \sqcap c_5$ and $c_1 \sqcup c_2$ when we compute them:

$$\begin{aligned} c_3 \sqcap c_5 &= (\{gibbons, humans\}, \{intelligent, thumbed\}) \\ &\quad \sqcap(\{cats, gibbons, dogs\}, \{hair\text{-}covered\}) \\ &= (\{gibbons\}, \sigma(\{gibbons\})) \\ &= (\{gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\ &= c_1 \end{aligned}$$

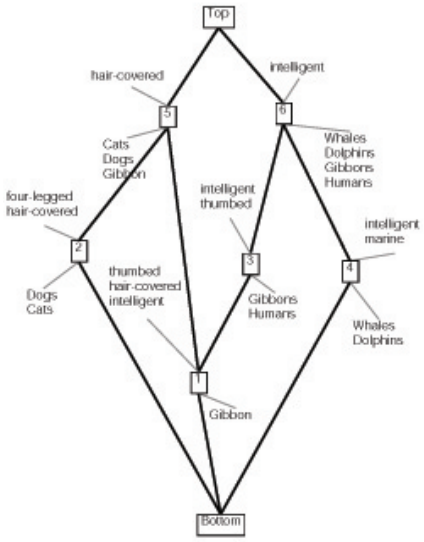

Figure B.2: The lattice of the mammals example with complete notation.

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

Generalizing, the fundamental theorem for concept lattices [Wil81] 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 [SR97], there is a restrictive definition based on subsets of superconcepts and subconcepts. They define *covers* and *subordinates*. A concept d *covers* concept c if $c \leq d$ and there is no concept e such that $c \leq e \leq d$. 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, meaning a subset of superconcepts of c.
Inversely, a concept c *subordinates* concept d if $d \leq c$ and there is no concept e such that $d \leq e \leq c$. If c subordinates d, we say "d is subordinated by c". The set of subordinates of concept c, denoted by *subordinates(c)*, is the set of concepts d such that c subordinates d, meaning a subset of subconcepts of c.

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:

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

The *top concept* reflects the properties that fit to *all* elements, and the *bottom concept* reflects the elements that fit to *all* properties. If there are not properties that fit all elements and/or there are not elements that fit all properties, the definitions are reduced to the following expressions:

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

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 $\mathcal{T}$.

B.4 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 we label with all the elements and properties calculated for the concept. The figure B.2 shows the lattice of the mammal example with this notation.
- $l(c) = (E_n, P_m)$, if c is the *largest* concept (w.r.t. $\leq$) with $p(\in P_m)$ in its intent, and c is the *smallest* concept (w.r.t $\leq$) 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 $\mu(e) = \bigwedge\{c \in \mathcal{L}(\mathcal{K}) \mid e \in ext(c)\}$. The figure B.1 shows the lattice of the mammal example with this notation.

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 all the lattice.

B.5 Concepts Builder Algorithms

There are several algorithms for computing the concepts for a given context, such as Siff et. al. [SR97], Godin et. al. [GMM+98] or Snelting et. al. [ST98]. We describe two main algorithms: a simple bottom-up one introduced in Siff et. al. [SR97] -just to understand easily how the concepts are built- and the most efficient one introduced in Ganter et. al. [GW99]. 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.

B.5.1 Bottom-up Algorithm

Given a set of elements $\mathcal{E}$, the smallest concept with extent containing $\mathcal{E}$ is $(\tau(\sigma(\mathcal{E})), \sigma(\mathcal{E}))$. 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. If we want to see the computation of the atomic concepts in the mammal example, the result 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 clear 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 worklist is formed containing all pairs of atomic concepts (*c', c*) such that $c \not\leq c'$ and $c' \not\leq c$. While the worklist is not empty, remove an element of the worklist (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 $c \not\leq c''$ and $c'' \not\leq c$ to the worklist. The process is repeated until the worklist is empty. The iterative process of the concept-building algorithm for our mammal examples is the following one:

$$\begin{aligned}
c_0 &= (\{cats, dogs\}, \{hair\text{-}covered, four\text{-}legged\}) \\
c_1 &= (\{gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\
c_2 &= (\{dolphins, whales\}, \{intelligent, marine\}) \\
c_3 &= (\{gibbons, humans\}, \{intelligent, thumbed\}) \\
Worklist &= [(c_0, c_1), (c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3)] \\
c_4 &= c_0 \sqcup c_1 = (\{cats, gibbons, dogs\}, \{hair\text{-}covered\}) \\
Worklist &= [(c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_2 &= \top = (\{cats, gibbons, dogs, dolphins, humans, whales\}, \emptyset) \\
Worklist &= [(c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_3 &= \top \\
Worklist &= [(c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_5 &= c_1 \sqcup c_2 = (\{gibbons, dolphins, humans, whales\}, \{intelligent\}) \\
Worklist &= [(c_2, c_3), (c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_3 &= c_5 \\
Worklist &= [(c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_4 &= \top \\
Worklist &= [(c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_3 \sqcup c_4 &= \top \\
Worklist &= [(c_0, c_5), (c_4, c_5)] \\
c_0 \sqcup c_5 &= \top \\
Worklist &= [(c_4, c_5)] \\
c_4 \sqcup c_5 &= \top \\
Worklist &= \emptyset
\end{aligned}$$

B.5.2 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* [GW99] 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 **lectically smaller** than a subset $B \neq A$ if the smallest element which distinguishes A and B belongs to B. Formally:

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

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 lectic order. If we have solved this, we can obviously generate all extents as follows: The lectically smallest concept extent is $\tau(\sigma(\emptyset))$. The other extents are found incrementally by determining the one which is lectically closest to the last extent found. In the end, we obtain the lectically largest extent, namely $\mathcal{E}$.

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

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

It is easy to verify the following statements:

1. $A < B \Leftrightarrow A <_i B$ for one $i \in \mathcal{E}$
2. $A <_i B$ and $A <_j C$ with $i < j \Rightarrow C <_i B$
3. $i \notin A \Rightarrow A < A \oplus i$
4. $A <_i B$ and B extent $\Rightarrow A \oplus i \subseteq B$, d.h. $A \oplus i \leq B$
5. $A <_i B$ and B extent $\Rightarrow A <_i A \oplus i$

The following theorem shows how we can find the concept extent we are looking for

Theorem 5. The smallest concept extent larger than a given set $A \subset \mathcal{E}$ (with respect to the lectic 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 lectically smallest extent is $\tau(\sigma(\emptyset))$. For a given set $A \subset \mathcal{E}$ we find the lectically 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 B.3 shows the application of the algorithm to calculate the extents. For 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*}.

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

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 B.3: Calculation of the extents of the mammal example using Ganter algorithm

B.6 Lattice Builder Algorithm

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*. Algorithm B.1 shows the simplest algorithm that ConAn uses to build the lattice.

Algorithm B.1: Algorithm to build the lattice.

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

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.

B.7 Concept Partitions

Given a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$, a **concept partition** is a set of concepts whose extents form a partition of $\mathcal{E}$. That is, $T = \{(X_0, Y_0), ...(X_{k-1}, Y_{k-1})\}$ is a *concept partition* iff the union of the extents of the concepts is the element set (i.e. $\bigcup X_i = \mathcal{E}$) and are pairwise disjoint ($X_i \bigcap X_j = \emptyset$ for $i \neq j$ and $X_i, X_j \in \mathcal{E}$).

An **atomic partition** of a concept lattice consisting of exactly the atomic concepts. A concept lattice need not have an atomic partition. For example, the lattice in Figure B.1 does not have an atomic partition. The atomic concepts are c_0, c_1, c_2 and c_3; however, c_1 and c_3 overlap - the element *gibbons* is in the extent of both concepts.

In order to develop tools to work with concept partitions, it is useful to be able to guarantee the existence of atomic partitions. Contexts that result in atomic concepts that, in turn, form a concept partition can be characterized precisely by the following definition: a context $(\mathcal{E}, \mathcal{P}, \mathcal{I})$ is *well-formed* if and only if, for every pair of elements $x, y \in \mathcal{E}$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$.

While not every context results in a concept lattice that has an atomic partition, we can *extend any context –by adding additional attributes–* to make it well-formed. Informally, a *context extension* is another context over the same set of elements (but a possibly augmented set of properties) whose concept lattice offers as least as many ways of grouping the elements as did the lattice derivable from the original context. More

formally, a context $(\mathcal{E}, \mathcal{P}', \mathcal{I}')$ is an extension of context $(\mathcal{E}, \mathcal{P}, \mathcal{I})$ if and only if $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{I} \subseteq \mathcal{I}'$.

There are several ways in which a non-well-formed context can be extended into a well-formed context. The important step in any such process is to identify the *offending* pairs of objects x and y for which $\sigma(\{x\}) \subsetneq \sigma(\{y\})$. This inequity may be counterbalanced by the addition of a property such that, in the resulting context, $\sigma(\{x\}) \not\subseteq \sigma(\{y\})$ Two such ways to extend a context to well-formed context are described below:

- A context can be extended via the addition of *unique identification properties*[3] for each pair of elements, x, y such that $\sigma(\{x\}) \subsetneq \sigma(\{y\})$, a new property p_x that uniquely identifies x is added to the extended property set. x becomes the *only* element that has the property p_x in the extended context relation (i.e., $\tau(\{p_x\}) = \{x\}$.

  As an example, consider the mammal context shown in Table B.1. The context is not well-formed because the properties of *human* are a proper subset of the properties of *gibbons*. To make a extension with *unique properties*, we augment the property set to include the property p_{human}. The resulting context is shown in the Table B.4. The resulting concept lattice is shown in Figure B.3 and the Table B.5 shows the extent and intent corresponding to the nodes in the lattice. Table B.6 shows the partitions of the lattice. Partition T_1 is the atomic partition

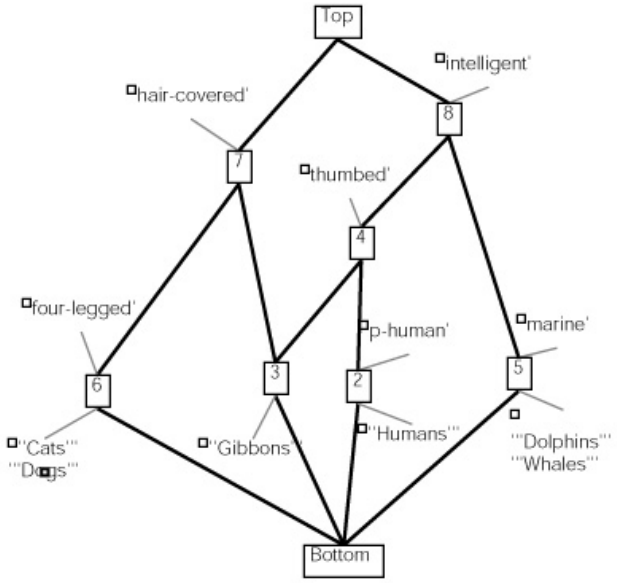

Figure B.3: The lattice for the mammals example with unique properties.

	four-legged	hair-covered	intelligent	marine	thumbed	p_{human}
Cats	×	×				
Dogs	×	×				
Dolphins			×	×		
Gibbons		×	×		×	
Humans			×		×	×
Whales			×	×		

Table B.4: The extension with unique properties of mammal context

[3] In [SR97], the use the name *unique identification attributes*. We changed it to keep our terminology of properties consistent

top	({ Cats, Gibbons, Dogs, Dolphins, Humans, Whales } , $\emptyset$)
c_8	({ Gibbons, Dolphins, Humans, Whales } , { intelligent })
c_7	({ Cats, Gibbons, Dogs } , { hair-covered })
c_6	({ Cats, Dogs } , { hair-covered, four-legged })
c_5	({ Dolphins, Whales } , { intelligent, marine })
c_4	({ Gibbons, Humans } , { intelligent, thumbed })
c_3	({ Gibbons } , { hair-covered, intelligent, thumbed })
c_2	({ Human }, {intelligent, thumbed, p_{human} })
bottom	($\emptyset$, { four-legged, hair-covered, intelligent, marine, thumbed, p_{human} })

Table B.5: Concepts of the mammal example

T_1	$\{c_2, c_3, c_5, c_6\}$
T_2	$\{c_2, c_5, c_7\}$
T_3	$\{c_4, c_5, c_6\}$
T_4	$\{c_6, c_8\}$
T_5	$\{top\}$

Table B.6: Concept partitions of the mammal concept extended with unique properties

- A context can be extended to a well-formed context by augmenting a context with *negative information*. Given a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$, a *complement* of a property $p \in \mathcal{P}$ is a property $\bar{p}$ such that $\tau(\{\bar{p}\}) = \{x \in \mathcal{E} | (x, a) \notin \mathcal{I}\}$. That is, $\bar{p}$ is an property of exactly the objects that do not have property p. The *complemented extension* of a context $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$ is a new context $(\mathcal{E}, \mathcal{P}', \mathcal{I}')$ formed by the algorithm introduced in Algorithm B.2:

  Algorithm B.2: Algorithm to calculate the complemented extension of a context
  (1) $\mathcal{P}' \leftarrow \mathcal{P}$
  (2) $\mathcal{I}' \leftarrow \mathcal{I}$
  (3) **while** $(\mathcal{E}, \mathcal{P}', \mathcal{I}')$ is not well formed do
  (4) let $x, y \in \mathcal{E}$ be such that $\sigma(\{x\}) \subsetneq \sigma(\{y\})$
  (5) let $p \in \mathcal{P}'$ be such that $a \notin \sigma(\{x\}), p \notin \sigma(\{y\})$
  (6) $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\bar{p}\}$, where $\bar{p}$ is a new property
  (7) $\mathcal{I}' \leftarrow \mathcal{I}' \cup \{(x, \bar{p}) | (x, p) \notin \mathcal{I}'\}$
  (8) **endwhile**

  For example, we can form a different well-formed extension of the mammal context shown in Table B.1 by creating the complemented extension. To make the complemented extension, we augment the property set to include the complementary property *not hair-covered*. The resulting context is shown in Table B.7. The resulting concept lattice is shown in B.4 and Table B.8 shows the intent and extent corresponding to the nodes in the lattice. Table B.9 shows the partitions of the lattice. Partition T_1 is the atomic partition.

It should be clear that both forms of extension result in well-formed contexts.

Both uniquely-attributed extensions and complemented extensions result in a concept lattice with at least as many (and frequently many more) nodes than the lattice derived from the original context. We say that a concept lattice $\mathcal{L}'$ derived from a $\mathcal{C}' = (\mathcal{E}, \mathcal{P}', \mathcal{I}')$ is an *extension* of a concept lattice $\mathcal{L}$ derived from a $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$ if $\mathcal{P} \subseteq \mathcal{P}'$, and for every concept c in $\mathcal{L}$, there is a concept c' in $\mathcal{L}'$ with the same extent. More formally, if $X \subseteq \mathcal{E}$ such that $\tau(\sigma(X)) = X$, then $\tau'(\sigma'(X)) = X$ where τ' and σ' are the common-element and common-property relations, respectively, for context C'.

Given a context C, both uniquely-attributed extensions and complemented extensions of $\mathcal{C}$ result in concept lattices that are the extensions of the lattice derived from C. In both cases, properties are added to the

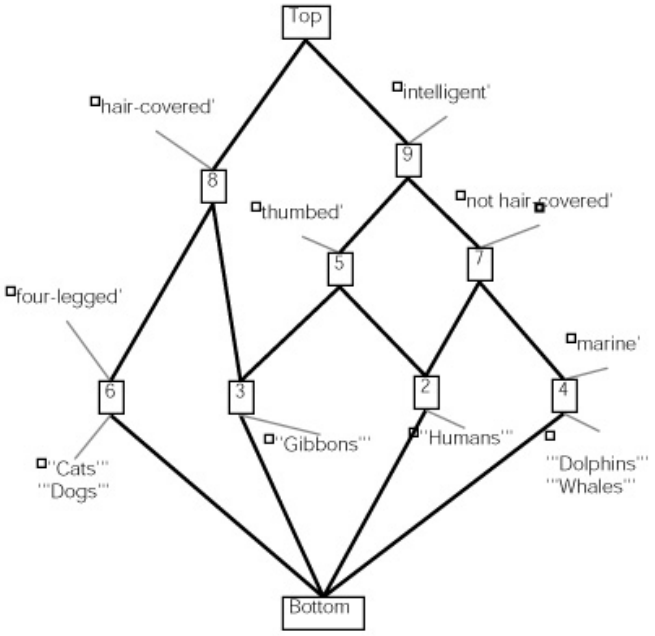

Figure B.4: The lattice of the complemented mammals example.

	four-legged	hair-covered	intelligent	marine	thumbed	not hair-covered
Cats	×	×				
Dogs	×	×				
Dolphins			×	×		×
Gibbons		×	×			
Humans			×		×	×
Whales			×	×		×

Table B.7: The extension with complemented extension mammal context

top	({ Cats, Gibbons, Dogs, Dolphins, Humans, Whales } , $\emptyset$)
c_9	({ Gibbons, Dolphins, Humans, Whales } , { intelligent })
c_8	({ Cats, Gibbons, Dogs } , { hair-covered })
c_7	({ Dolphins, Human, Whales }, {not hair-covered })
c_6	({ Cats, Dogs } , { hair-covered, four-legged })
c_5	({ Gibbons, Humans } , { intelligent, thumbed })
c_4	({ Dolphins, Whales } , { intelligent, marine, not hair-covered })
c_3	({ Gibbons } , { hair-covered, intelligent, thumbed })
c_2	({ Human }, {intelligent, thumbed, not hair-covered })
bottom	($\emptyset$, { four-legged, hair-covered, intelligent, marine, thumbed, not hair-covered })

Table B.8: Concepts of the mammal example

T_1	$\{c_2, c_3, c_4, c_6\}$
T_2	$\{c_2, c_4, c_8\}$
T_3	$\{c_3, c_6, c_7\}$
T_4	$\{c_4, c_5, c_6\}$
T_5	$\{c_7, c_8\}$
T_5	$\{top\}$

Table B.9: Concept partitions of the mammal concept extended with complemented properties

extended context to make in easier to distinguish elements. For example, if $\sigma\{x\} \subsetneq \sigma\{y\}$ then there is at least one property which is a property of y that is not a property of x. Whether adding unique attributes or complementary attributes, negative information is represented in a positive form in the extended context to help distinguish such x and y.

B.8 Finding Partitions in a Concept Lattice

We say that concept lattice derived from a well-formed context is a well-formed concept lattice. Once the context is well-formed, we are able to find the different partitions with the algorithm introduced in Algorithm B.3.

Algorithm B.3: Algorithm to find the partitions of a well-formed concept lattice.

```
(1)   A ← covs(⊥)
(2)   T ← {A}
(3)   W ← {A}
(4)   while W ≠ ∅ do
(5)       remove some t from W
(6)       for each c ∈ t
(7)           for each c' ∈ covers(c)
(8)               t' ← t − subs(c')
(9)               if (∪t') ∩ c' = ∅
(10)                  t'' ← t' ∪ {c'}
(11)                  if t'' ∉ T
(12)                      T ← T ∪ {t''}
(13)                      W ← W ∪ {t''}
(14)                  endif
(15)              endif
(16)          endfor
(17)      endfor
(18)  endwhile
```

The algorithm builds up a collection of all the partitions of a concept lattice. Let T the collection of partitions we are forming. Let W be a worklist of partitions. We begin with the atomic partitions, which is the *covers* of the *bottom* concept of the concept lattice. T and W are both initialized to the singleton set containing the atomic partition.

The algorithm works by considering partitions from Worklist W until W is empty. For each partition removed from W, new partitions are formed (when possible by selecting a concept of the partition, choosing a cover of that concept, adding it to the partition, and removing overlapping concepts.

As an example, consider the atomic partition of the concept lattice derived from the uniquely-attributed mammal context (B.3). The algorithm begins with the atomic partition (consisting of concept c_2, c_3, c_5 and c_6) as the sole member of the worklist. The algorithm removes the atomic partition from the worklist, as t in line (5). Suppose that in the first iteration of the for loop in line (6), c refers to c_6. the covering set of c_6

is the singleton set consisting of c_7, so c' is assigned c_7 in line (7). In line (8), t' is assigned the value of p minus the subordinate concepts of c_7 (*i.e.*, c_3, c_6 and $bottom$, so t' is $\{c_2, c_5\}$). In line (9), the union of the extents of c_2 and c_5 is disjoint with the extent of c_7; thus, in line (10), the partition $p'' = \{c_5, c_2\} \cup \{c_7\}$ is formed. p'' is added to the set of partitions and to the worklist in line (12) and line (13).

In the worst case, the number of partitions can be exponential in the number of concepts. Furthermore, the techniques for making contexts well-formed, discussed so far, only exacerbate the problem: More precise means of distinguishing sets of elements translates to more concepts. This is turns leads to more possibilities for partitions.

If the number of concepts in a concept lattice is large, it may be impractical to consider every possible partitions of the concepts. In such a case, it is possible to adapt the algorithm to work interactively, with guidance from the user. Before attempting to find a new partition, the algorithm would pause for the user to specify *seed sets* of concepts , which would be used to force the algorithm to find only coarser partitions than the seed sets (i.e., partitions that do not subdivide the seed sets).

Bibliography

[AACGJ01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *Proceedings of ASE '01 (16th Conference on Automated Software Engineering)*, pages 166–173. IEEE Computer Society Press, November 2001.

[AB84] Mark S. Aldenderfer and Roger K. Blashfield. *Cluster Analysis*. Sage University Paper Series on Quantitative Applications in the Social Sciences. Sage Publications Inc., Beverly Hills, 1984.

[ABN04] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting Implicit Collaboration Patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

[ADN03] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. X-Ray Views: Understanding the Internals of Classes. In *Proceedings of ASE '03 (18th Conference on Automated Software Engineering)*, pages 267–270. IEEE Computer Society Press, October 2003. Short paper.

[Aeb03] Tobias Aebi. Extracting Architectural Information using Different Levels of Collaboration. Diploma Thesis, University of Bern, September 2003.

[AL97] Nicolas Anquetil and Timothy Lethbridge. File Clustering using Naming Conventions for Legacy Systems. In *Proceedings of CASCON '97*, pages 184–195, November 1997.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 150 Louis Pasteur, University of Ottawa, Ottawa, Canada, 1999.

[And73] M. R. Andenberg. *Cluster Analysis for Applications*. Academic Press, London, 1973.

[Aré03] Gabriela Arévalo. Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis. In *Proceedings of LMO '03 (Langages et Modeles à Objets)*, pages 47–59. Hermes, Paris, January 2003.

[AT01] Paul Anderson and Tim Teitelbaum. Software Inspection Using CodeSurfer. In *Proceedings of WISE'01 (International Workshop on Inspection in Software Engineering)*, 2001.

[AYLCB96] Sihem Amer-Yahia, Lotfi Lakhal, Rosine Cicchetti, and Jean Paul Bordat. iO2 - An Algorithmic Method for Building Inheritance Graphs in Object Database Design. In *Proceedings of ER'96 (15th International Conference on Conceptual Modeling)*, volume 1157 of *Lecture Notes in Computer Science*, pages 422–437. Springer-Verlag, 1996.

[Bal99] Thomas Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BE81] Laszlo. A. Belady and Carlo J. Evangelisti. System Partitioning and its Measure. *Journal of Systems and Software*, 2:23–29, 1981.

[BH65] G. H. Ball and D. J. Hall. Isodata, A Novel Method of Data Analysis and Pattern Classification. Technical report, Stanford Research Institute, Menlo Park, California, 1965.

[Bir40] Garret Birkhoff. Lattice Theory. *American Mathematical Society*, 1940.

[BM70] M. Barbut and B. Monjardet. *Ordre et Classification*. Hachette, 1970.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley Press, 1996.

[Bro96] Kyle Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters Thesis, North Carolina State University, 1996.

[BS91] Rodrigo A. Botafogo and Ben Shneiderman. Identifying Aggregates in Hypertext Structures. In *Proceedings CHI '91 (Conference on Human Factors in Computing Systems)*, pages 63–74. ACM Press, 1991.

[BSR+01] Kathrin Böttger, Rolf Schwitter, Debbie Richards, Oscar Aguilera, and Diego Mollá. Reconciling Use Cases via Controlled Language and Graphical Models. In *Proceedings of INAP '01 (14th International Conference on Applications of Prolog)*, pages 20–22, Japan, October 2001. University of Tokyo.

[Buc02] Frank Buchli. An explicit model for ADVance. Informatikprojekt, University of Bern, December 2002.

[Buc03] Frank Buchli. Detecting Software Patterns using Formal Concept Analysis. Diploma thesis, University of Bern, September 2003.

[Bud91] Timothy A. Budd. *An Introduction to Object-Oriented Programming*. Addison Wesley, 1991.

[BV00] Dragan Bojic and Dusan Velasevic. Reverse Engineering of Use Case Realizations in UML. In *Proceedings of SAC '00 (ACM Symposium on Applied Computing)*. ACM Press, 2000.

[CC92] Elliot J. Chikofsky and James H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[CCLL99] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *Proceedings of IWPC '99 (7th International Workshop on Program Comprehension)*, pages 136–143. IEEE, IEEE Computer Society, May 1999.

[CIMH98] James H. Cross II, Saeed Maghsoodloo, and Dean Hendrix. Control Structure Diagrams: Overview and Evaluation. *Journal of Empirical Software Engineering*, 3(2):131–158, 1998.

[CL96] Jan-Bon Chen and Samuel C. Lee. Generation and Reorganization of Subtype hierarchies. *Journal of Object Oriented Programming*, 8(8):26–35, January 1996.

[CLR90] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Coo92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 27, pages 1 – 15. ACM Press, October 1992.

[CS90] Song C. Choi and Walt Scacchi. Extracting and Restructuring the Design of Large Systems. *IEEE Software*, pages 66–71, January 1990.

[DDHL95] Hervé Dicky, Christoph Dony, Marianne Huchard, and Thérèse Libourel. ARES, Adding a class and REStructuring Inhertitance Hierarchy. In *BDA : 11èmes Journées Bases de Données Avancées*, pages 25–42, 1995.

[DDHL96] Hervé Dicky, Christoph Dony, Marianne Huchard, and Thérèse Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications)*, pages 251–267. ACM Press, 1996.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Dek03] Uri Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, February 2003.

[DGLD04] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Reengineering Environments*. 2004. to appear.

[DH98] Stephan Düwel and Wolfgang 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.

[DH00] Stephan Düwel and Wolfgang 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.

[DHHaV04] Michel Dao, Marianne Huchard, Mohamed Rouane Hacene, and Cyril Roume andPetko Valtchev. Improving Generalization Level in UML Models Iterative Cross Generalization in Practice. In *Proceedings of ICCS '94 (12th International Conference on Conceptual Structures)*, volume 3127 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, July 2004.

[DHL$^+$02] Michel Dao, Marianne Huchard, Thérèse Libourel, Cyril Roume, and Hervé Leblanc. A New Approach to Factorization: Introducing Metrics. In *Proceedings of METRICS '02 (8_{th} IEEE International Symposium on Software Metrics*, pages 227 – 236. IEEE Computer Society, 2002.

[DP02] B.A. Davey and H. A. Priestley. *Introduction to Lattices and Order: Second Edition*. Cambridge University Press, 2002.

[DRW00] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[Düw99] Stephan Düwel. Enhancing System Analysis by Means of Formal Concept Analysis. In *Conference on Advanced Information Systems Engineering. 6th Doctoral Consortium*, Heidelberg, Germany, June 1999.

[EKS01a] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of ICSM '01 (International Conference on Software Maintenance)*. IEEE Computer Society Press, 2001.

[EKS01b] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-Driven Program Understanding using Concept Analysis of Execution Traces. In *Proceedings of IWPC '01 (9th International Workshop on Program Comprehension)*, pages 300–309. IEEE Computer Society Press, 2001.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, March 2003.

[Eve74] B. Everitt. *Cluster Analysis*. Heineman Educational Books, London, 1974.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fis98] Bernd Fischer. Specification-based Browsing of Software Component Libraries. In *Proceedings of ASE '98(13th Conference on Automated Software Engineering*, pages 74–83. IEEE Computer Society Press, 1998.

[FLS95] Petra Funk, Anke Lewien, and Gregor Snelting. Algorithms for Concept Lattice Decomposition and their Application. Technical Report 95-09, Computer Science Dept., Technische Universitat Braunschweig, 1995.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[Fur86] George W. Furnas. Generalized Fisheye View. In *Proceedings of CHI '86 (Conference on Human Factors in Computing Systems)*, pages 16–23. ACM Press, 1986.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[GHRV02] Robert Godin, Marianne Huchard, Cyrill Roume, and Petko Valtchev. Inheritance and Automation: Where Are We Now? In Andrew Black, Erik Ernst, Peter Grogono, and Markky Sakkinen, editors, *ECOOP 2002: Proceedings of the Inheritance Workshop*, pages 58–64. University of Jyväskylä, June 2002.

[GL70] I. Gitman and M. D. Levine. An Algorithm for Detecting Unimodal Fuzzy Sets and Its Application as a Clustering Technique. *IEEE Transactions on Computers*, CE-19:583–593, July 1970.

[GL91] Keith Brian Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. *Transactions on Software Engineering*, 17(18):751–761, August 1991.

[GM93] Robert Godin and Hafedh Mili. Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices. In *Proceedings OOPSLA '93 (8th Conference on Object-Oriented Programming Systems, Languages, and Applications)*, volume 28, pages 394–410, October 1993.

[GMM+95] Robert Godin, Guy Mineau, Rokia Missaoui, Marc St-Germain, and Najib Faraj. Applying Concept Formation Methods to Software Reuse. *International Journal of Knowledge Engineering and Software Engineering*, 5(1):119–142, 1995.

[GMM+98] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[Gor81] A. D. Gordon. *Classification: Methods for the Exploratory Analysis of Multivariate Data*. Chapman & Hall Ltd., London, 1981.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[HB85] David H. Hutchens and Victor R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.

[HCIM02] Dean Hendrix, James H. Cross II, and Saeed Maghsoodloo. The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities. *IEEE Transactions on Software Engineering*, 28(5):463–477, May 2002.

[HDL00] Marianne Huchard, Hervé Dicky, and Hervé Leblanc. Galois Lattice as a Framework to specify Algorithms Building Class Hierarchies. *Theoretical Informatics and Applications*, 34:521–548, 2000.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concept and Techniques*. Morgan Kaufmann, 2000.

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

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, 1988.

[JGR99] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[JR97] Dean Jerding and Spencer Rugaber. Using Visualization for Architectural Localization and Extraction. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 56 – 65. IEEE Computer Society Press, 1997.

[KG88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace – Understanding Object-Oriented Systems using Concurrently Animated Views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, November 1988.

[KM00] Tobias Kuipers and Leon Moonen. Types and Concept Analysis for Legacy Systems. Technical Report SEN-R0017, Centrum voor Wiskunde en Informatica, July 2000.

[KO01] Sergei Kuznetsov and Sergei Obëdkov. Comparing Performance of Algorithms for Generating Concept Lattices. In *Proceedings of International Workshop on Concept Lattice-based Theory, Methods and Tools for Knowledge Discovery in Databases*, 2001.

[Kos00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[KP96] Christian Kramer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of WCRE '96 (3rd Working Conference on Reverse Engineering)*, pages 208–216. IEEE Computer Society Press, November 1996.

[KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons Inc., New York, 1990.

[KS94] Maren Krone and Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of ICSE '94 (16th International Conference on Software Engineering)*, pages 49–57. IEEE Computer Society / ACM Press, 1994.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-Based Reverse Engineering of Design Components. In *Proceedings of ICSE '99 (21st International Conference on Software Engineering)*, pages 226–235. IEEE Computer Society Press / ACM Press, May 1999.

[KST96] Edward J. Klimas, Suzanne Skublics, and David A. Thomas. *Smalltalk with Style*. Prentice-Hall, 1996.

[Lan99] Michele Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma Thesis, University of Bern, October 1999.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[LB85] Manny M. Lehman and Les Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.

[LD01] Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[Leb00] Hervé Leblanc. *Sous-hiérarchies de Galois: un Modèle pour la Construction et L'évolution des Hiérarchies d'objets (Galois Sub-hierarchies: a Model for Construction and Evolution of Object Hierarchies)*. PhD thesis, Université Montpellier 2, 2000.

[Lin95] Christian 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.

[LN95] Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 342–357. ACM Press, 1995.

[LPLS96] David Littman, Jeannine Pinto, Stan Letovsky, and Elliot Soloway. Mental Models and Software Maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98. Ablex Publishing Corporation, 1996.

[LRP95] John Lamping, Ramana Rao, and Peter Pirolli. A Focus + Context Technique based on Hyperbolic Geometry for Visualising Large Hierarchies. In *Proceedings of CHI '95 (International Conference on Human Factors in Computing Systems)*. ACM Press, 1995.

[Lun98] Chung-Horng Lung. Software Architecture Recovery and Restructuring through Clustering Techniques. In *Proceedings of the 3rd International Workshop on Software Architecture*, pages 101–104. ACM Press, 1998.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[Mac67] J. B. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Symposium on Mathematics, Statistics and Probability*, pages 281–297, Berkley, 1967. University of California Press.

[MH00] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 2nd edition, 2000.

[MM98] Spiros Mancoridis and Brian S. Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[MM01] Brian S. Mitchell and Spiros Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements. In *Proceedings of ICSM '01 (International Conference on Software Maintenance)*. IEEE Computer Society Press, November 2001.

[MMCG99] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[Moo96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications)*, pages 235–250. ACM Press, 1996.

[MS89] Michele Missikoff and Michel Scholl. An algorithm for insertion into a lattice: Application to type classification. In *Proceedings of FODO '89 (3rd International Conference on Foundations of Data Organization and Algorithms*, volume 367 of *Lecture Notes in Computer Science*, pages 64–82. Springer-Verlag, June 1989.

[MS95] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software — Concepts and Tools*, 16:170–182, 1995.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of ECOOP'98 (European Conference on Object-Oriented Programming)*, number 1445 in Lecture Notes in Computer Science, pages 355–383. Springer-Verlag, 1998.

[MU90] Hausi A. Müller and James S. Uhl. Composing Subsystem Structures using (k,2)-partite graphs. In *Proceedings of ICSM '90 (International Conference on Software Maintenance)*, pages 12–19. IEEE Computer Society Press, November 1990.

[Mül86] Hausi A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[Mül93] Hausi A. Müller. Software Engineering Education should concentrate on Software Evolution. In *Proceedings of National Workshop on Software Engineering Education*, pages 102 – 104, May 1993. University of Victoria (Canada).

[Nei96] James M. Neighbors. Finding Reusable Software Components in Large Systems. In *Proceedings of WCRE '96 (3rd Working Conference on Reverse Engineering)*, pages 2–10. IEEE Computer Society Press, November 1996.

[NR89] Jakob Nielsen and John T. Richards. The Experience of Learning and Using Smalltalk. *IEEE Software*, 6(3):73–77, 1989.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jürg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of ICSE '02 (24th International Conference on Software Engineering)*, pages 338–348. ACM Press, 2002.

[OMG99] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.

[Par94] David Lorge Parnas. Software Aging. In *Proceedings of ICSE '94 (International Conference on Software Engineering)*, pages 279–287. IEEE Computer Society / ACM Press, 1994.

[RB02a] Debbie Richards and Kathrin Böttger. Assisting Decision Making in Requirements Reconciliation. In *Proceedings of CSCWD '02 (7th International Conference on Computer Supported Cooperative Work in Design)*, Brazil, September 2002. IEEE Computer Society Press.

[RB02b] Debbie Richards and Kathrin Böttger. A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices. In *Proceedings of AI '02 (15th Australian Joint Conference on Artificial Intelligence)*, number 2557 in LNAI, pages 579 – 590, Canberra, Australia, 2002. Springer Verlag.

[RB02c] Debbie Richards and Kathrin Böttger. Representing Requirements in Natural Language as Concept Lattices. In *Proceedings of the 22nd Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (ES2002)*, Cambridge, December 2002. Springer Verlag.

[RBF02a] Debbie Richards, Kathrin Böttger, and Anne Fure. RECOCASE-tool: A CASE Tool for RECOnciling Requirements Viewpoints. In *Proceedings of AWRE '02 (7th Australian Workshop on Requirements Engineering)*, 2002.

[RBF02b] Debbie Richards, Kathrin Böttger, and Anne Fure. Using RECOCASE to compare use cases from multiple viewpoints. In *Proceedings of ACIS '02 (13th Australasian Conference on Information Systems)*, Melbourne, Australia, December 2002.

[RHM00] Peter Rosner, Tracy Hall, and Tobias Mayer. Measuring Object-Orientedness: The Invocation Profile. In Reiner R. Dumke and Alain Abran, editors, *Proceedings of IWSM '00 (10th International Workshop on New Approaches in Software Measurement)*, pages 18–28. Springer-Verlag, October 2000.

[Ric02] Tamar Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Berne, May 2002.

[Riv96] Fred Rivard. Smalltalk : a Reflective Language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[Rog71] J. H. Roger. Updating a minimum spanning tree. algorithm as40. *Applied Statistics*, 20:204–206, 1971.

[Ros69] G. J. S. Ross. Minimum spanning tree. algorithm as13. *Applied Statistics*, 18:103–104, 1969.

[Rou02] Cyrill Roume. Évaluation Structurelle de la Factorisation et la Généralisation au sein des Hiérarchies de Classes: Introduction de Métriques. *L'Objet*, 8(1-2):151–166, 2002.

[Run92] Elke A. Rundensteiner. A Class Classification Algorithm for Supporting Consistent Object Views. Technical report, University of Michigan, 1992.

[SAP89] Robert W. Schwanke, Rita Z. Altucher, and Michael A. Platoff. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of International Workshop on Software Specification and Design*, pages 147–150. IEEE Computer Society Press, 1989.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[SFM99] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems*, 44:171–185, 1999.

[Sha96] Subhash Sharma. *Applied Multivariate Techniques*. John Wiley, 1996.

[SLL+88] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing Documentation to Compensate for Delocalized Plans. *Commununications of ACM*, 31(11):1259–1267, 1988.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 268–285. ACM Press, 1996.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275 – 284. IEEE Computer Society Press, 1995.

[SMLD97] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and Francois Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *Proceedings of ASE '97 (12th International Conference on Automated Software Engineering)*, pages 210–218. IEEE, IEEE Computer Society Press, November 1997.

[SMSP04] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A Scalable Approach to User-Session based Testing of Web Applications through Concept Analysis. In *Proceedings of ASE '04 (19th International Conference on Automated Software Engineering)*, pages 132–141. IEEE Computer Society Press, September 2004.

[Sne96] Gregor Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[Sne98] Gregor 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. ACM Press.

[SP89] Robert W. Schwanke and Michael A. Platoff. Cross References are Features. In *Proceedings of Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[Spi89] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[SR97] Michael Siff and Thomas Reps. Identifying Modules via Concept Analysis. In *Proceedings of ICSM '97 (International Conference on Software Maintenance)*, pages 170–179. IEEE Computer Society Press, 1997.

[SRMK99] Reinhard Schauer, Sébastian Robitaille, Francois Martel, and Rudolf Keller. Hot-Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*. IEEE Computer Society Press, 1999.

[SS73] P.H.A. Sneath and R.R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. W. H. Freeman and Company, San Francisco, 1973.

[ST97] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. Technical 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.

[ST98] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[Ste92] Robert A. Stegwee. *Division for Conquest - Decision Support for Information Architecture Specification*. Ph.D. thesis, University of Groningen, Groningen, The Netherlands, 1992.

[SvG98] Jochen Seemann and Jürgen Wolff 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.

[TA99] Paolo Tonella and Giuliano Antoniol. Object Oriented Design Pattern Inference. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 230–238. IEEE Computer Society Press, October 1999.

[TC04] Paolo Tonella and Mariano Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *Proceedings of WCRE 2004 (11th International Working Conference in Reverse Engineering)*, pages 112–121. IEEE Computer Society Press, November 2004.

[TCBE03] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In Gerd Stumme, editor, *Proceedings of ICFCA '03 (1st International Conference on Formal Concept Analysis)*. Springer-Verlag, February 2003.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings of ISPSE '00 (International Conference on Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.

[TH98] Vassilios Tzerpos and R. C. Holt. Software Botryology, Automatic Clustering of Software Systems. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*. IEEE Computer Society Press, 1998.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[Til03] Thomas 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.

[Ton01] Paolo Tonella. Concept Analysis for Module Restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, April 2001.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying Objects using Cluster and Concept Analysis. In *Proceedings of ICSE '99 (21st International Conference on Software Engineering)*, pages 246–255. ACM Press, 1999.

[Vis03] Cincom Smalltalk, September 2003. http://www.cincom.com/scripts/smalltalk.dll/.

[vL93] Gregor von Laszewski. A Collection of Graph Partitioning Algorithms: Simulated Annealing, Simulated Tempering, Kernighan Lin, Two Optimal, Graph Reduction, Bisection. Technical Report SCCS 477, Northeast Parallel Architectures Center, Syracuse University, Syracuse, New York, 1993.

[vR79] C. v. Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[WH92] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[Wig97] Theo Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 33 – 43. IEEE Computer Society Press, 1997.

[Wil81] Rudolf Wille. Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, 83:445–470, September 1981.

[Wis69] David Wishart. Mode Analysis: A Generalization of Nearest Neighbour which reduces chaining effects. *Numerical Taxonomy*, 1969.

[XLZS04] Xia Xu, Chung-Horng Lung, Maria Zaman, and Anand Srinivasan. Program Restructure through Clustering Technique. In *Proceedings of International Workshop on Source Code Analysis and Manipulation*, pages 75 – 84. IEEE, IEEE Computer Society Press, September 2004.

Curriculum Vitae

Personal Information

Name:	Gabriela Beatriz Arévalo
Nationality:	Argentinian
Date of Birth:	September 14^{th}, 1973
Place of Birth:	San Salvador de Jujuy, Argentina

Education

Sept. 2000 - Jan. 2005:	Ph.D. in Computer Science in the Software Composition Group, University of Bern, Switzerland Subject of the Ph.D. thesis: *"High-Level Views in Object-Oriented Systems using Formal Concept Analysis"*
Sept. 1999 - Aug. 2000:	Master of Science in Computer Science in the context of EMOOSE (European Master in Object-Oriented Software Engineering) program in Ecole des Mines de Nantes, France. Subject of the Master thesis: *"Object-Oriented Architectural Description of Frameworks"* Degree awarded by Faculteit Van de Wetenschappen, Vrije Universiteit Brussel, Belgium
1992 - Mar. 1999:	Licenciatura en Informática in Facultad de Ciencias Exactas in Universidad Nacional de La Plata, Buenos Aires, Argentina Subject of Licenciatura thesis: *"G.I.S + Oceans = A strange combination"* (in Spanish)
1987 - 1991:	Secondary School in Buenos Aires, Argentina
1979 - 1986:	Primary School in Buenos Aires, Argentina