Analysing Object-Oriented Application Frameworks Using Concept Analysis ⋆

Gabriela Arévalo[2] and Tom Mens[1]

[1] Postdoctoral Fellow of the Fund for Scientific Research - Flanders
Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be
[2] Software Composition Group, University of Berne
Bern, Switzerland
arevalo@iam.unibe.ch

Abstract. This paper proposes to use the formal technique of *Concept Analysis* to analyse how methods and classes in an object-oriented inheritance hierarchy are coupled by means of the *inheritance* and *interfaces* relationships. Especially for large and complex inheritance hierarchies, we believe that a formal analysis of how behaviour is reused can provide insight in how the hierarchy was built and the different relationships among the classes. To perform this analysis, we use behavioural information provided by the *self sends* and *super sends* made in each class of the hierarchy. The proposed technique allows us to identify weak spots in the inheritance hierarchy that may be improved, and to serve as guidelines for extending or customising an object-oriented application framework. As a first step, this paper reports on an initial experiment with the *Magnitude* hierarchy in the *Smalltalk* programming language.

1 Introduction

Understanding a software application implies to know how the different entities are related. In the case of an object-oriented application framework, our entities are classes and methods. When a developer defines a class in an application, he requires knowledge about how behaviour and structure have to be reused using inheritance techniques. It is not trivial to achieve optimal reuse, especially when the number of classes is large or the inheritance hierarchy is deep. In these situations, *concept analysis* can be used as a technique to help us cope with these problems, by analysing the inheritance and interface relationships among the classes in the class hierarchy. Then we can understand and document the way inheritance is used in the framework, and use this information to provide guidelines for how the framework can be modified or customised without running into behavioural problems or without breaching the design conventions used when building the framework.

Concept Analysis (CA) is a branch of lattice theory that allows us to identify meaningful groupings of *elements* (referred to as *objects* in CA literature) that have common

⋆ In Advances in Object Oriented Information System: OOIS 2002 Workshops, J.M. Bruel and Z. Bellahsene (Eds.), pp. 53-63, Springer Verlag, 2002

properties (referred to as *attributes* in CA literature) [1]. These groupings are called *concepts* and capture similarities among a set of *elements* based on their common *properties*. Mathematically, concepts are *maximal collections of elements sharing common properties*. They form a complete partial order, called a *concept lattice*, which represents the relationships between all the concepts [1, 15, 6]. To use the CA technique, one only needs to specify the properties of interest on each element, and does not need to think about all possible combination of these properties, since these groupings are made automatically by the CA algorithm.

2 Applying concept analysis to inheritance hierarchies

In this paper, we report on an experiment that uses concept analysis to analyse an existing inheritance hierarchy with the aim to better understand how inheritance is used in practice to achieve reuse, and to provide guidelines to improve the inheritance hierarchy. To achieve this, we analyse classes and their methods based on their relationships in terms of *inheritance*, *interfaces* and *message sending behaviour*. The *inheritance relationship* indicates whether a class is an ancestor or descendant of another one. The *interface relationship* indicates which methods are defined abstract or concrete in each class. The *message sending behaviour* indicates which methods are called by other methods in a class. Because we are mainly interested in reuse of behaviour, we will only look at *self sends* and *super sends*.

As a first step, we need to define the *elements* and *properties* we wish to reason about to apply the CA technique. Because we are interested in classes in an object-oriented hierarchy, together with their methods and the messages sent by these methods, we define an *element* as a pair (C, s) such that "a method with signature s is called (via a self send or super send) by some method implemented in the class C". For the CA properties, we chose a classification based on the relationships explained previously:

Classification based on message sending behaviour. (C, s) satisfies predicate ***calledViaSelf*** if s is called via a self send by some method in C. (C, s) satisfies predicate ***calledViaSuper*** if s is called via a super send by some method in C.

Classification based on interface relationship. (C, s) satisfies predicate ***isConcreteIn:*** D if s is implemented as a concrete method in class D. (C, s) satisfies predicate ***isAbstractIn:*** D if s is implemented as an abstract method in class D.

Classification based on inheritance relationship (C, s) satisfies predicate ***isDefinedInAncestor:*** D if D defines s and is an ancestor class (i.e., a direct or indirect superclass) of C. (C, s) satisfies predicate ***isDefinedInDescendant:*** D if D defines s and is a descendant class (i.e., a direct or indirect subclass) of C. (C, s) satisfies predicate ***isDefinedLocally*** if C defines s. This means that s is defined in the same class that calls it.

[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.

CA properties are then defined as conjunctions obtained by taking one predicate from each classification. Below, we present some of the properties that can be obtained by a conjunction of the predicates presented previously.

- Predicate ***concreteSuperCaptureIn:*** D is a conjunction of *calledViaSuper*, *isConcreteIn:* D and *isDefinedInAncestor:* D. (C, s) satisfies this predicate if s is called via a super send in some method of C, and the receiver method is implemented in the class D that is an ancestor class of C.
- Predicate ***concreteSelfCaptureLocally:*** C is a conjunction of *calledViaSelf*, *isConcreteIn:* C and *isDefinedLocally*. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the same class C.
- Predicate ***concreteSelfCaptureInAncestor:*** D is a conjunction of *calledViaSelf*, *isConcreteIn:* D and *isDefinedInAncestor:* D. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the class D that is an ancestor class of C.
- Predicate ***concreteSelfCaptureInDescendant:*** D is a conjunction of *calledViaSelf*, *isConcreteIn:* D and *isDefinedInDescendant:* D. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the class D that is a descendant class of C.
- Predicate ***abstractSelfCaptureLocally:*** C is a conjunction of *calledViaSelf*, *isAbstractIn:* C and *isDefinedLocally*. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as an abstract one in the same class C.

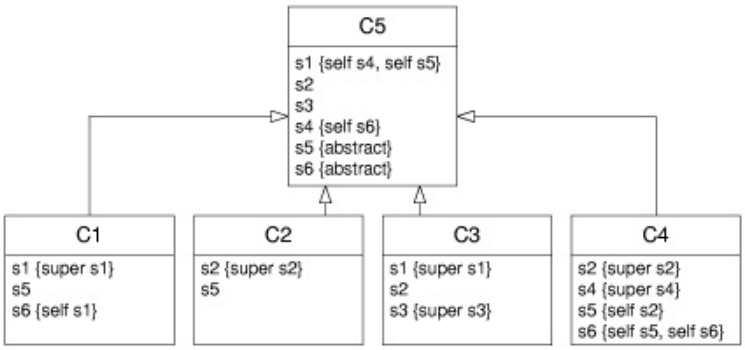

Fig. 1. Example class hierarchy

As an example of how these properties can be used to compute *concepts*, take a look at the example class hierarchy of Figure 1. All self sends and super sends in the source code have been annotated between curly braces. Based on this information, the CA algorithm will automatically compute the following concepts (among others):

- **Concept 1** has elements { (C_4, s_2), (C_4, s_5), (C_4, s_6) } and properties { *concreteSelfCaptureLocally:* C_4 }. This means that only the selectors s_2, s_5 and s_6 are called via a *self* send that is captured by *concrete* method implementations in the class C_4 itself.
- **Concept 2** has elements { (C_1, s_1), (C_2, s_2), (C_3, s_1), (C_3, s_3), (C_4, s_2), (C_4, s_4) } and properties { *concreteSuperCaptureIn:* C_5 }. This means that only the selectors s_1, s_2, s_3 and s_4 are called via a *super* send in the classes C_1, C_2, C_3, C_4 and they are implemented by a *concrete* method in the ancestor C_5 of these classes.
- **Concept 3** has elements { *(*C_5, s_5*), (*C_5, s_6*)* } and properties { *abstractSelfCaptureLocally:* C_5*, concreteSelfCaptureInDescendant:* C_1*, concreteSelfCaptureInDescendant:* C_4*,* }. This means that abstract methods s_5 and s_6 in C_5 are defined concrete in the subclasses C_1 and C_4.

In the remainder of this paper, we will make the concept notation more compact by grouping together all selectors that belong to the same class. For example, the element set of **concept 2** can be abbreviated to {(C_1,s_1), (C_2,s_2), (C_3,{s_1, s_3}), (C_4,{s_2, s_4})}. We will also identify each concept by a unique number that is automatically assigned to the concept by the CA algorithm.

3 Case study

The abstract example of section 2 was only intended to make the reader understand how the process works. Our actual experiment consists of applying the CA technique to study the *Magnitude* inheritance hierarchy of Smalltalk in more detail[2]. We decided to use the *Magnitude* hierarchy for our first experiment because: it is sufficiently large to get meaningful results (29 classes, 894 methods); it heavily relies on code reuse by inheritance (19 abstract methods, 296 self sends, 49 super sends); it is stable and well-documented; it is commonly available for most versions of Smalltalk. Figure 2 displays the part of the *Magnitude* hierarchy that is used for the examples later in this paper.

Based on results provided by the CA algorithm, we analyzed the relationships between the classes in terms of *inheritance*, *interface* and *message sending behaviour*. With SOUL, a logic meta-programming language built on top of –and tightly integrated with– Smalltalk [17], we extracted 248 elements and 73 properties. [3] Based on this, the CA algorithm that we implemented directly in Smalltalk computed 125 concepts as a result.

As we said previously, the properties will be of the form *concreteSuperCaptureIn: C, concreteSelfCaptureInDescendant: C, abstractSelfCaptureLocally: C, concreteSelfCaptureLocally: C, concreteSelfCaptureInAncestor: C, ...*. If we abstract the argument *C* out of these properties, we find that many concepts resemble each other because they contain the same set of properties. This commonality between concepts allows us to

[2] For our experiments, we worked in VisualWorks release 5i4, and restricted ourselves to only those classes belonging to the Smalltalk namespaces Core, Graphics, Kernel, and UI.

[3] For our experiment we computed a static approximation of the self sends and super sends. For example, even if sends occur in a conditional branch that is never executed, they are still extracted by our algorithm.

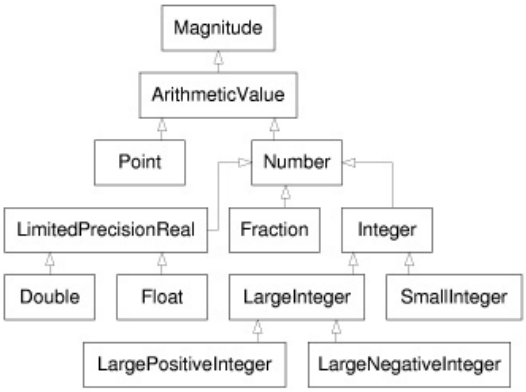

Fig. 2. Smalltalk *Magnitude* class hierarchy

identify *concept patterns*. A *concept pattern* consists of a textual and graphical description, a concrete example related to the *Magnitude* class hierarchy, and an analysis of how the pattern provides more insight in how parts of the code are reused.

Concept pattern 1: Self sends captured locally

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in a class B and they are implemented in the same class. Figure 3 shows this concept pattern graphically. It occurs in 21 concepts of the *Magnitude* concept lattice.

For example, **concept 71** has elements {*(Fraction,* {*reduced, negative, asFloat, asDouble*})} and properties {*concreteSelfCaptureLocally: Fraction*}.

This concept pattern is useful to document the *internal interface* of a class, i.e., the set of all selectors that are implemented in the class and to which self sends are made by methods implemented in the same class. This internal interface captures and documents the core behaviour of the class. This can be used to distinguish the core methods of a class from the auxiliary ones. This is important information for reusers because, if the core methods are overridden in a subclass, all auxiliary methods will still work correctly with the new core [10].

Concept pattern 2: Self sends captured in ancestor

A set of selectors $m_1 \ \ldots \ m_p$ implemented in a class B are called via *self* send in descendant classes $A_1 \ \ldots \ A_n$. Figure 4 displays this concept pattern graphically. It occurs in 9 concepts of the *Magnitude* concept lattice.

For example, **concept 73** has elements {*(LargePositiveInteger,* {*digitLength, digitAt:*}*), (LargeNegativeInteger,* {*digitLength, digitAt:*})} and properties {*concreteSelfCaptureInAncestor: LargeInteger*}.

This concept pattern is useful to detect the *actual subclass interface* of a class, i.e., the set of all selectors that are implemented in the class and to which self sends are made

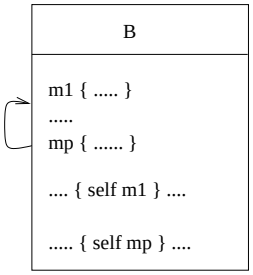

Fig. 3. Concept Pattern 1

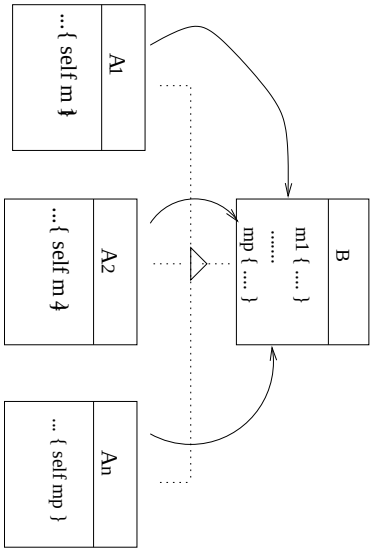

Fig. 4. Concept Pattern 2

by subclasses. Changes to these methods will also have an impact in all subclasses that reuse it. For example, using **concept 73** we know that if we change the implementation of *digitLength* or *digitAt:* in *LargeInteger*, we have to check whether the methods in *LargePositiveInteger* and *LargeNegativeInteger* that call these methods still behave as expected.

In terms of software refactoring [4], the concept pattern can sometimes be used to identify common code in sibling classes that is useful to refactor in the common superclass. For example, **concept 73** illustrates that, to a certain extent, sibling classes *LargePositiveInteger* and *LargeNegativeInteger* reuse the behaviour defined in their superclass *LargeInteger* in the same way. Further investigation of the actual source code allows us to discover that the self sends (digitLength: and digitAt:) are invoked from within the implementation of the method *compressed* in both sibling classes and the implementation of this method is very similar in both cases. Hence, a refactoring might be appropriate to extract this common behaviour into an auxiliary method that can be pulled up into the common superclass *LargeInteger*. This analysis showed us a limitation of our approach: we should not only take the receiver of a self send into account (in this case *digitLength* and *digitAt:*) but also the sender (in this case *compressed*), since this represents essential information.

Concept pattern 3: Super call

A set of selectors $m_1 \ldots m_p$ implemented in the class B are called via a *super* send in descendant classes $A_1 \ldots A_n$. Figure 5 illustrates this concept pattern graphically. It occurs in 8 concepts of the *Magnitude* concept lattice.

For example, **concept 105** has elements {*(Float,* $\{>, \geq, \leq\}$*), (Double,* $\{>, \geq, \leq\}$*), (SmallInteger,* $\{>, \geq, \leq\}$*), (LargeInteger,* $\{>, \geq, \leq\}$*)*} and properties {*concreteSuperCaptureIn: Magnitude*}.

This concept pattern can be used to detect the *actual overriding interface* of a class, i.e., the set of all selectors that are implemented in the class and to which super sends are made by methods implemented in descendants. For example, **concept 105** shows that $\{>, \geq, \leq\}$ is an important part of the overriding interface of *Magnitude*, since each of these selectors are overridden in descendant classes *Float*, *Double*, *SmallInteger* and *LargeInteger* for optimisation purposes.

The concept pattern can also detect situations of *implementation inheritance* [11]. Typically, when implementation inheritance is used, a class overrides many methods defined in its parent (and uses super sends to invoke the parent behaviour). Finally, the concept pattern can provide guidelines for framework customisation. If we define a new subclass of a given class, it is likely that we have to override the methods specified in the overriding interface of the parent class. For example, if we would decide to create a new subclass of *LimitedPrecisionReal* or *Integer*, it is very likely that we need to override all the selectors in $\{>, \geq, \leq\}$.

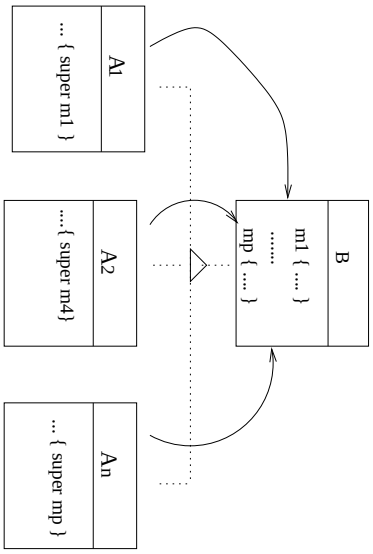

Fig. 5. Concept Pattern 3

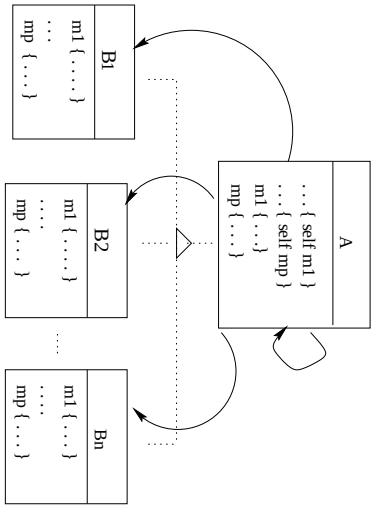

Fig. 6. Concept Pattern 4

Concept pattern 4: Local self send captured in descendants

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in the class A and the selectors are implemented in A and in some of its descendant classes $B_1 \ldots B_k$. Figure 6 shows

this concept pattern in a general way. It occurs in 31 concepts of the *Magnitude* concept lattice.

For example, **concept 69** has elements {*(Number,* {*raisedTo:, sqrt, ln, truncated*})} and properties {*concreteSelfCaptureInDescendant: Float, concreteSelfCaptureInDescendant: Double, concreteSelfCaptureLocally: Number*}.

This concept pattern documents which specific methods are overridden in the subclasses of a common superclass. This means that the superclass defines some common or default behaviour for these methods, and each of the descendants can override this implementation via the mechanism of late binding with subclass-specific behaviour.

Concept pattern 5: Local self send with super delegation

A set of selectors $m_1 \ldots m_p$ are called via a *self* and *super* send in a class A and the selectors are implemented in the same class A as well as in an ancestor class B. Figure 7 illustrates this concept pattern graphically. It occurs in 4 concepts of the *Magnitude* concept lattice. For example, **concept 48** has elements {*(SmallInteger,* {$>$, $\geq$, $\leq$})} and properties {*(concreteSuperCapture: Magnitude, concreteSelfCaptureLocally: SmallInteger*}

This concept pattern documents delegation between methods in the same class and with the superclass. In all the found cases, the method that calls a selector via a *super send* has the same name as the selector itself. For example, in *SmallInteger* the method $\geq$ contains a "`super` $\geq$" statement. This means that part of the action to be executed (when a *self send* is made) is defined in the superclass, and the message is delegated by a *super send*.

Concept pattern 6: Template methods and hook methods

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in a class A and the selectors are implemented as abstract methods in the same class A and are implemented as concrete methods in descendant classes $B_1 \ldots B_k$. Figure 8 illustrates this concept pattern graphically. It occurs in 7 concepts of the *Magnitude* concept lattice.

In the *Magnitude* hierarchy, this concept pattern only occurs for the subhierarchies with root classes *Integer* and *ArithmeticValue*. For example, **concept 31** has elements {*(ArithmeticValue,* {*, -})} and properties {*abstractSelfCaptureLocally: ArithmeticValue, concreteSelfCaptureInDescendant:* {*LargeInteger, Fraction, Integer, SmallInteger, Float, FixedPoint, Point, Double*} }.

In this example, the abstract methods {*,-} in *ArithmeticValue* are called by other methods of the same class, but the actual implementation is defined in descendant classes. This concept pattern identifies the *hot spots* in an object-oriented application framework [9, 3]. These hot spots are implemented by means of so-called *template methods* and *hook methods* [16, 5]. In their simplest form, template methods are methods that perform self sends to abstract methods, which are the hook methods that are expected to be overridden in subclasses.

The information expressed in this concept pattern identifies the *abstract interface* of a class, as well as the subclasses that provide a concrete implementation of this

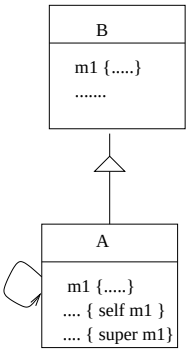

Fig. 7. C. Pattern 5

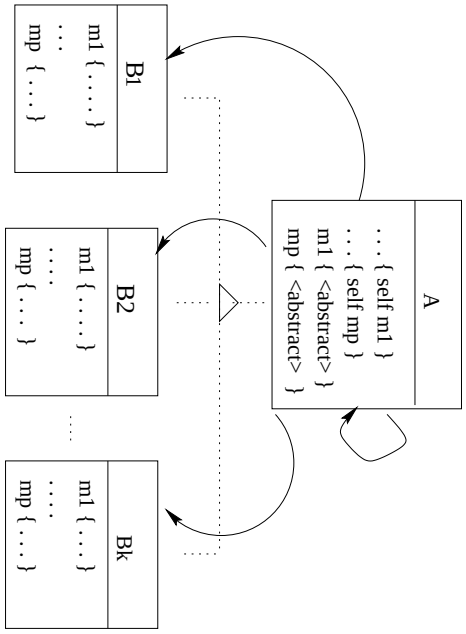

Fig. 8. Concept Pattern 6

interface. This information is essential during framework customisation when we want to add a *concrete* subclass of an *abstract* class, because it tells us which methods should be at least be implemented.

4 Related work

Godin and Mili [7, 8] used concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. They showed how Cook's [2] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. In C++ and Java, Snelting and Tip [13] analysed a class hierarchy by making the relationship between methods 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. The approach proved useful to serve as a basis for automated or interactive restructuring tools for class hierarchies. Siff and Reps [12] used concept analysis to modularise legacy C programs into C++ classes. Last but not least, Tonella and Antoniol [14] used concept analysis to infer structural design patterns from C++ code, which also provides crucial information to get a deeper understanding of object-oriented application frameworks.

All the above approaches only took information into account about which selectors are implemented by which classes. More behavioural information (e.g., based on self and super sends) was not considered. Hence, they could only detect *interface inheritance* but not *implementation inheritance*. As shown in this paper, more behavioural

information about how a subclass is derived from its subclass is essential to analyse and understand the kind of reuse that is achieved.

5 Conclusion and Future Work

In this paper we analysed the well-known *Magnitude* inheritance hierarchy in Smalltalk using Concept Analysis. Based on information about self sends, super sends and invoked methods, we calculated the concept lattice for this hierarchy. We classified the generated concepts into *concept patterns*, which provide a roadmap of the code that ought to be analysed and understood. With the information given by the concept patterns, we discovered a number of interesting non-documented relationships about how classes and methods in the hierarchy are reused. A preliminary analysis of these patterns strengthened our belief that the technique is useful to: document the subclass interface of a class; provide guidelines on how an object-oriented framework can be customised or reused; identify hot spots in an object-oriented application framework; detect the type of inheritance (e.g. interface inheritance or implementation inheritance) used in an inheritance hierarchy; identify opportunities for refactoring; get insight in the potential impact of changes to framework classes. Based on these results, we believe that Concept Analysis is a promising technique in the understanding and re-engineering of large inheritance hierarchies.

Based on these results, we know that a lot of further research is necessary. One research avenue concerns the applicability of CA. We intend to confirm the usefulness of our method by analysing other well-known and non-trivial Smalltalk class hierarchies (e.g., Collection, Model, View and Controller). We also want to apply our approach to other object-oriented languages (such as Java and C++) to investigate the effect of language-specific properties (such as interfaces or multiple inheritance) by comparing similar class hierarchies in different languages. Another topic of future work is to investigate the effect of other behavioural information such as method invocations, variable accesses and variable updates; or the effect of other essential relationships between classes, such as composition and aggregation. Finally, we should take into account the additional information provided by how the concepts in the generated concept lattice are related via a partial order.

6 Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project ”Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02). We thank Dirk Deridder, Oscar Nierstrasz, Roel Wuyts and the referees for their feedback.

References

1. G. Birkhoff. Lattice theory. *American Mathematical Society*, 1940.

2. W. R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15. ACM Press, October 1992.
3. S. Demeyer. Analysis of overriden methods to infer hot spots. In S. Demeyer and J. Bosch, editors, *ECOOP '98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
4. M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.
6. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
7. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proc. Int'l Conf. Oject-Oriented Programs, Systems, Languages and Applications*, volume 28 of *ACM SIGPLAN Notices*, pages 394–410. ACM Press, October 1993.
8. R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.
9. R. E. Johnson and B. Foote. Designing reusable classes. *J. Object-Oriented Programming*, 1(2):22–35, Feb. 1988.
10. J. Lamping. Typing the specialization interface. In *Proc. Int'l Conf. Oject-Oriented Programs, Systems, Languages and Applications*, volume 28 of *ACM SIGPLAN Notices*, pages 201–214. ACM Press, October 1993.
11. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
12. M. Siff and T. Reps. Identifying modules via concept analysis. In *Proc. Int. Conf. Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.
13. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *ACM Trans. Programming Languages and Systems*, 1998.
14. P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proc. Int'l Conf. Software Maintenance*, pages 230–238. IEEE Computer Society Press, 1999.
15. R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, pages 445–470, September 1981.
16. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.
17. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. Int'l Conf. TOOLS USA'98*, pages 112–124. IEEE Computer Society Press, 1998.