

---

# Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis

**Gabriela Arévalo**

*Software Composition Group  
University of Berne  
Bern, Switzerland  
arevalo@iam.unibe.ch*

---

*ABSTRACT. The functionalities of software artifacts are defined by structural and behavioral dependencies. During evolution and maintenance phases of a system, the developer has to be able to understand how these dependencies were defined and how they influence the interaction of the artifacts. The developer must be sure that modifications done in the system will not break its behavior. In the most of the cases, this happens because the dependencies are not documented . We propose to tackle this problem in the context of object oriented classes hierarchies using Concept Analysis. We use different properties about invocations in methods to analyze the dependencies among the hierarchy classes in terms of class behaviour. Based on these results, we show a set of patterns that describe repeated kinds of behavior in class hierarchies. We show the application of these patterns in the specific case of Magnitude hierarchy in Smalltalk*

*RÉSUMÉ. Les fonctionnalités des objets sont définies par des dépendances structurales et comportementales. Pendant les phases d'évolution et d'entretien d'un système, le programmeur doit comprendre comment ces dépendances ont été définies et comment elles influencent l'interaction des objets. Le programmeur doit s'assurer que les modifications faites au système ne changeront pas son comportement. Dans la plupart des cas, ceci se produit parce que les dépendances ne sont pas documentées. Nous proposons d'aborder ce problème dans le contexte des hiérarchies de classes orientées objets en utilisant l'analyse de concepts (concept analysis). Nous employons différentes propriétés des invocations dans les méthodes pour analyser les dépendances parmi les hiérarchie des classes en termes du comportement de classes. Basé sur ces résultats, nous montrons un ensemble de patrons qui décrivent des répétitions de comportements dans hiérarchies. Nous montrons l'application de ces patrons dans le cas spécifique de la hiérarchie Magnitude de Smalltalk*

*KEYWORDS: concept analysis, class hierarchy, program understanding*

*MOTS-CLÉS : analyse de concepts, hiérarchie des classes, compréhension de programs*

---

## 1. Introduction

A system is composed of software artifacts and its functionality is defined by their interaction. Generally, the interaction is constrained by structural and behavioral dependencies which define the relationships between the artifacts. During evolution and maintenance phases of a system, the developer has to understand how these dependencies are defined and how they influence the interaction of the artifacts.

In object-oriented applications, one important mechanism is inheritance. The classes in a hierarchy have dependencies in terms of structural and behavioural aspects. The superclass/subclass relationships provide the developer with information about the structural dependencies. Behavioral dependency is introduced when the classes define new methods, override existing methods and reuse the behaviour provided by the parent class. Several activities make a hierarchy evolve: defining new class, adding new behaviour to existing classes, identifying abstract classes using concrete ones, identifying interfaces in the classes, etc. Clearly, evolution and maintenance tasks add and modify the dependencies between the classes [LUC 97]. Most of the cases the dependencies are not explicit and the non-rational modification of the hierarchy can lead to inadequate inheritance structures, missing abstractions, overly specialized components or deficient object modelling [CAS 95, MIK 98].

To solve the problem of non-explicit (evolved) design constraints, in our approach we propose to identify the behavioural dependencies among the classes using inferred relationships provided by concept analysis. Using as our main elements the *self-send* and *super-send* invocations in the methods, we define properties based on *inheritance* and *interface* relationships between the classes. Then we run CA, and we identify maximal groups of elements and properties that allows us detect repeated kinds of behaviour in a hierarchy. We call them as *concept patterns*. These patterns help us to understand how the hierarchy was built. We show the application of these patterns in the particular case of *Magnitude* hierarchy in Smalltalk.

This paper is organized as follows: Section 2 introduces the minimal definitions needed to understand how concept analysis is used. Section 3 explains how we use this technique to understand class hierarchies. Section 4 explains which were case studies and some identified *concept patterns*. Section 5 provides some guidelines about the lessons learned with this application of concept analysis for class hierarchies analysis. Finally, section 6 and 7 summarizes the related work and conclusion and future work.

## 2. Concept Analysis - Main Concepts

Concept analysis (CA) [GAN 99] is a branch of lattice theory (also known as Galois lattices [BAR 70, WIL 81]) that allows us to identify meaningful groupings of *elements* (referred to as *objects* in CA literature) that have common *properties* (re-

ferred to as *attributes* in CA literature)<sup>1</sup>. To illustrate concept analysis, we consider an easy example which shows which kinds of music the people prefer. The *elements* are a group of people *Frank, Anne, Arthur, John, Thomas, Michael*; and the *properties* are *Rock, Pop, Jazz, Folk, Tango*<sup>2</sup>. Table 1 shows which people prefer which kind of music.

| <i>prefers</i> | Rock | Pop  | Jazz | Folk | Tango |
|----------------|------|------|------|------|-------|
| Frank          | True | True |      | True |       |
| Anne           | True | True |      |      | True  |
| Arthur         |      |      | True | True |       |
| Catherine      |      |      | True |      |       |
| Thomas         |      |      | True |      |       |
| Michael        |      |      | True | True |       |

**Table 1.** *Elements and their satisfied properties in the Music example*

In order to understand the basics of concept analysis, a few definitions are required. A *context* is a triple  $C=(E, P, R)$ , where  $E$  and  $P$  are finite sets (elements and properties, respectively), and  $R$  is a binary relation between  $E$  and  $P$ . In the example about music, the elements are the people, the properties are the different kinds of music. The binary relation  $R$  is given in Table 1. For example, the tuple  $(Frank, Folk)$  is in  $R$ , but  $(Anne, Jazz)$  is not.

Let  $X \subseteq E$  and  $Y \subseteq P$ . The mappings

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

which gives us all the *common attributes* of the elements contained in  $X$ , and

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

which gives us the *common objects* of the properties contained in  $Y$ , form a *Galois lattice*.

With both definitions, we can define what is the unit of concept analysis: the *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  $Y = \sigma(X)$  and  $X = \tau(Y)$ . That is, a concept is a maximal collection of objects sharing common attributes. In the example,  $(\{Frank, Anne\}, \{Rock, Pop\})$  is a concept, whereas  $(\{Catherine\}, \{Jazz\})$  is not a concept: Although  $\sigma(\{Catherine\}) = \{Jazz\}$ ,  $\tau(\{Jazz\}) = \{Arthur, Catherine, Thomas, Michael\}$ . A key indicating the extent and intent of each concept is shown in the table 2.

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

2. The complete property name is *prefers Pop* or *prefers Folk*. We abbreviate this information to make the reading as easy as possible

|        |  |
|--------|--|
| top    | ({ all elements }, $\emptyset$ )               |
| $c_7$  | ({Arthur, Catherine, Thomas, Michael}, {Jazz}) |
| $c_6$  | ({Frank, Arthur, Michael}, {Folk})             |
| $c_5$  | ({Frank, Anne}, {Rock, Pop})                   |
| $c_4$  | ({Arthur, Michael}, {Jazz, Folk})              |
| $c_3$  | ({Frank}, {Rock, Pop, Folk})                   |
| $c_2$  | ({Anne}, {Rock, Pop, Tango})                   |
| bottom | ( $\emptyset$ , { all properties })            |

**Table 2.** *The set of concepts of the example about Music*

There is a complete partial order which forms a *concept lattice* over the set of concepts. There are several algorithms for computing the concepts and the concept lattice for a given context ([KUZ 01], [LIN 00]). As this is only a summary of the main concepts needed to understand the use of concept analysis technique in our approach, the reader interested in a more deep explanations should refer to [GAN 99].

### 3. Representing Information for Understanding of Class Hierarchies

We propose to analyse classes and their methods based on their relationships of *inheritance* and *interfaces* using *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 exported by the classes. The *message sending behaviour* indicates which methods are called by other methods in a class. The behavioral dependencies are defined when we add, modify, override, or remove a method in a class. This kind of (evolution) activities can have an impact in the set of superclass and subclasses of the modified class. This impact is detected when there are *self* or *super* sends invocations to the changed methods of a class, because it shows us the reuse of superclass/subclass behavior of a given class at code-level -without having to take care of running a typing mechanism. This is the feature of the OO hierarchies we will analyze. As we are mainly interested in this specific case of (evolved) class behaviour, we will only look at *self sends* and *super sends*.

The CA technique requires us to define the *elements* and *properties* that we wish to reason about. Because we are interested in classes in an object-oriented hierarchy, together with their methods and the messages sent by these methods, we define each CA element as an invocation represented as (*Class, method, selector*) such that “*selector* is called (via a self send or super send) in the *method* defined in the *Class*”. For the CA properties, we chose a characterisation based on the following classification of predicates:

## Classification of the sender

- (*Class, method, selector*) satisfies the predicate ***calledViaSelf***, if *selector* is called via a self send in *method* in *Class*
- (*Class, method, selector*) satisfies the predicate ***calledViaSuper***, if *selector* is called via a super send in *method* in *Class*

## Classification of the Definition

- (*Class, method, selector*) satisfies the predicate ***isDefinedAsConcrete***: *otherClass*, if *selector* is defined as a concrete method in *otherClass*
- (*Class, method, selector*) satisfies the predicate ***isDefinedAsAbstract***: *otherClass*, if *selector* is defined as an abstract method in *otherClass*

## Classification of the (inheritance) relationship between sender and implementor classes

- (*Class, method, selector*) satisfies the predicate ***isDefinedInAncestor***: *ancestorClass*, if *ancestorClass* is an ancestor class (i.e., a direct or indirect superclass) of *Class* that defines *selector*.
- (*Class, method, selector*) satisfies the predicate ***isDefinedInDescendant***: *descendantClass*, if *descendantClass* is a descendant class (i.e., a direct or indirect subclass) of *Class* that defines *selector*.
- (*Class, method, selector*) satisfies the predicate ***isDefinedLocally***, if *Class* defines *selector*. This means that there is a definition of *selector* in the same class that calls it.

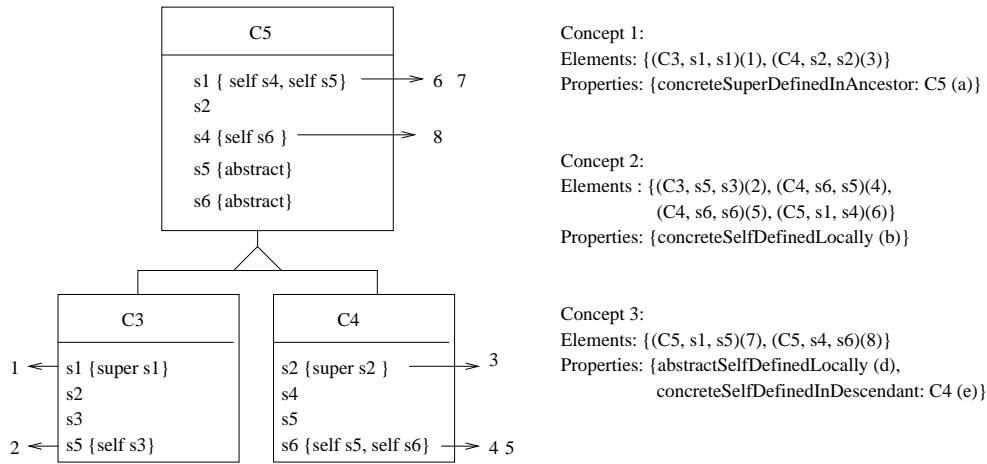
CA properties are then defined as conjunctions obtained by taking one predicate from each subgroup. Making all the possible combinations we get 12 CA properties, but we only mention those that appeared in our case studies. Supposing we have the element (*Class, method, selector*), the possible properties that fulfill are:

- Property ***concreteSuperDefinedInAncestor***: *ancestorClass* is a conjunction of the three predicates ***calledViaSuper*** and ***isDefinedAsConcreteIn***: *ancestorClass* and ***isDefinedInAncestor***: *ancestorClass*. This means that *selector* is called via super in *method*, that the *selector* is defined as a *concrete* method in *ancestorClass* and that *ancestorClass* is an ancestor of *Class*.
- Property ***concreteSelfDefinedLocally*** is a conjunction of the three predicates ***calledViaSelf*** and ***isDefinedAsConcreteIn***: *Class* and ***isDefinedLocally***. This means that *selector* is called via self in *method*, that the *selector* is defined as a *concrete* method in *Class* and this class is the same as where *selector* is invoked.
- Property ***concreteSelfDefinedInAncestor***: *ancestorClass* is a conjunction of the three predicates ***calledViaSelf*** and ***isDefinedAsConcreteIn***: *ancestorClass* and ***isDefinedInAncestor***: *ancestorClass*. This means that *selector* is called via self in *method*, that the *selector* is defined as a *concrete* method in *ancestorClass* and that *ancestorClass* is an ancestor of *Class*.

- Property **concreteSelfDefinedInDescendant**: *descendantClass* is a conjunction of the three predicates *calledViaSelf* and *isDefinedAsConcreteIn*: *descendantClass* and *isDefinedInDescendant*: *descendantClass*. This means that *selector* is called via self in *method*, that the *selector* is defined as a *concrete* method in *descendantClass* and that *descendantClass* is a descendant of *Class*.

- Property **abstractSelfDefinedLocally** is a conjunction of the three predicates *calledViaSelf* and *isDefinedAsAbstractIn*: *Class* and *isDefinedLocally*. This means that *selector* is called via self in *method*, that the *selector* is defined as an *abstract* method in *Class* and that *ancestorClass* and this class is the same as where *selector* is invoked.

As before, we use a boolean table to summarise which properties (specified in the columns) are specified by which elements (specified in the rows). An example of such a table is given in Table 3. The table represents the information provided in the left part of figure 1. Using the information present in this table, the CA technique computes the *concepts*. Some of the concepts are shown in the right part of the figure 1<sup>3</sup>.



**Figure 1.** Class hierarchy with invocations

Based on the information given by the concepts, we analyze the dependencies between the classes in terms of their defined methods and calls to their superclass/subclasses. The main issue is to propose some guidelines to understand how the classes were built and evolved during their lifecycle.

3. The indexes and the letters associated to the elements and properties are only used to help the reading of the concepts in the table

|                      | a    | b    | c    | d    | e    | f    |
|----------------------|------|------|------|------|------|------|
| 1. $(C_3, s_1, s_1)$ | True |      |      |      |      |      |
| 2. $(C_3, s_5, s_3)$ |      | True |      |      |      |      |
| 3. $(C_4, s_2, s_2)$ | True |      |      |      |      |      |
| 4. $(C_4, s_6, s_5)$ |      | True | True |      |      |      |
| 5. $(C_4, s_6, s_6)$ |      | True | True |      |      |      |
| 6. $(C_5, s_1, s_4)$ |      | True |      |      | True |      |
| 7. $(C_5, s_1, s_5)$ |      |      |      | True | True | True |
| 8. $(C_5, s_4, s_6)$ |      |      |      | True | True |      |

**Table 3.** Elements and their satisfied properties in an inheritance hierarchy. The attributes are: a: *concreteSuperDefinedInAncestor*:  $C_5$ , b: *concreteSelfDefinedLocally*, c: *abstractSelfDefinedInAncestor*:  $C_5$ , d: *abstractSelfDefinedLocally*, e: *concreteSelfDefinedInDescendant*:  $C_3$ , f: *concreteSelfDefinedInDescendant*:  $C_4$

#### 4. Case Study and Validation

The abstract example explained in the previous section was only intended to make the reader understand how the process works. In this section, we present the case studies which we have analysed and how the results are classified in *concept patterns*

##### 4.1. Experimental Setup

Our actual experiment consists of applying CA to study the following inheritance hierarchies of Smalltalk: *Magnitude* (47 classes), *Collection* (95 classes), *Model* (517 classes), *View* (95 classes) and *Controller* (73 classes), which are predefined ones in the Smalltalk environment. We chose them because we consider that they are stable enough, they have a large number of classes that are difficult to analyze manually and they are commonly available for most versions of Smalltalk. The elements (in our case, invocations) and the properties (explained in the previous section) have been calculated using Moose [DUC 00] (a language independent environment for reengineering object oriented systems). The concepts and the lattice were built using ConAn tool, which is our tool implemented in Smalltalk and used to calculate concepts and concept lattice, and uses the result of the information provided by Moose.

##### 4.2. Results: Concept Patterns

Based on the example presented previously, we see that the properties will be *concreteSuperDefinedInAncestor*:  $C$ , *concreteSelfDefinedInDescendant*:  $C$ , *abstractSelfDefinedLocally*, *concreteSelfDefinedLocally*, *concreteSelfDefinedInAncestor*:  $C$ , . . . . If we abstract the argument  $C$  out of these properties, we find that many concepts re-

|                                 |     |
|---------------------------------|-----|
| Classes                         | 29  |
| (Concrete and Abstract) Methods | 894 |
| Self calls                      | 296 |
| Super calls                     | 49  |
| CA elements (invocations)       | 248 |
| CA properties                   | 73  |
| Used Concepts                   | 80  |
| Total Concepts                  | 125 |

**Table 4.** Statistical information about experiments with Smalltalk Magnitude class hierarchy

semble each other because they contain the same set of properties. This commonality between concepts allows us to identify *concept patterns* (similar to the idea of Design Patterns [GAM 94]). A *concept pattern* consists of a textual and graphical description, one example related to a studied hierarchy, and an analysis of how the pattern provides more insight in how parts of the code are reused<sup>4</sup>.

We will show only some of the identified *concept patterns* using as example the *Magnitude* hierarchy. A list of all the concept patterns is in [AR02]. Table 4 provides the reader with some statistical information about the specific case of *Magnitude* hierarchy<sup>5</sup>. The *Used number of concepts* indicates how many concepts contain some useful information for our patterns. The *Total number of concepts* indicates how many concepts in total are generated by concept analysis. The difference between these two items (125 - 80) gives us the number of concepts which have combination of properties with no meaning at all in the context of class hierarchies.

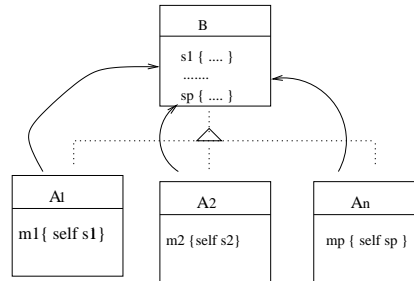
#### Concept pattern 1: Self sends defined in ancestor

The concept contains a set of selectors  $s_1 \dots s_p$  defined in a class  $B$  that are called via *self send* in the methods  $m_1 \dots m_p$  in descendant classes  $A_1 \dots A_n$ . This means that the intent of the concepts is composed of the property *concreteSelfDefinedInAncestor: B*. Figure 2 displays this concept pattern graphically.

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 by subclasses. Changes to these methods will also have an impact in all subclasses that reuse them. For example, in the analysis *Magnitude* hierarchy we have a concept with the elements  $\{(LargePositiveInteger, \{compressed, highBit\} dig-$

4. To make the notation for concepts more compact, we will group together all selectors belonging to the same class and all the methods that call the same selector. For example if we have in a concept with the elements  $\{(C_1, m_1, s_1), (C_1, m_2, s_1), (C_1, m_3, s_3), (C_2, m_4, s_4)\}$ , we will show it as  $\{(C_1, \{m_1, m_2\}, s_1), (m_3, s_3)\}, (C_2, m_4, s_4)\}$

5. 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



**Figure 2.** Concept pattern 1

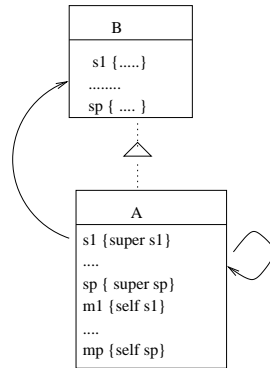
*itLength*}, {compressed, **digitAt**:}), (*LargeNegativeInteger*, {compressed, **digitLength**}, {compressed, **digitAt**:}) and properties {*concreteSelfDefinedInAncestor*: *Integer*}. Then we know that if we change the implementation of *digitLength* or *digitAt*: in *LargeInteger*, we have to check whether the methods in *LargePositiveInteger* and *LargeNegativeInteger* that call these methods still behave as expected.

In terms of software refactoring [FOW 99], the concept pattern can also be used to identify common code in sibling classes that is useful to refactor in the common superclass. For example, to a certain extent, sibling classes *LargePositiveInteger* and *LargeNegativeInteger* reuse the behaviour defined in their superclass *LargeInteger* in the same way invoking from within the implementation of the method *compressed* 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*. [FOW 99] calls this action as *Pull up method*.

### Concept pattern 2: Local self send with super delegation

A set of selectors  $s_1 \dots s_p$  are called via a *self* and *super* send in the methods  $m_1 \dots m_p$  in a class *A* and the selectors are defined in the same class *A* as well as in an ancestor class *B*. This means that the intent of the concepts is composed of the properties *concreteSelfDefinedLocally*, *concreteSuperDefinedInAncestor*: *B*. Figure 3 illustrates this concept pattern graphically.

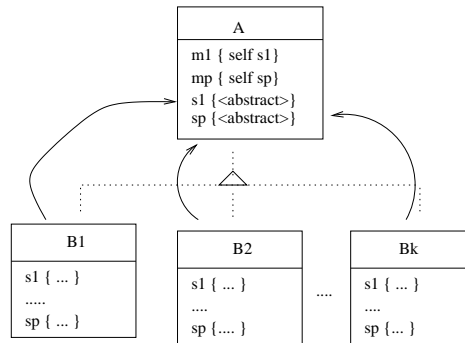
This concept pattern documents delegation between methods in the same class and with the superclass. For example, in the *Magnitude* hierarchy we have a concept with the elements.  $\{(SmallInteger, \{ \{ >, digitLength \}, > \}, \{ \{ \geq, printDigitsOn:base:, highBitAux: \}, \geq \}, \{ \{ \leq, digitLength \}, \leq \})\}$  and properties  $\{(concreteSuperDefinedInAncestor: Magnitude, concreteSelfDefinedLocally: SmallInteger)\}$ . In all the found cases, the method that calls a selector via a *super send* has the same name as the selector itself. 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*.



**Figure 3.** *Concept pattern 2*

**Concept pattern 3: Template methods and hook methods**

A set of selectors  $s_1 \dots s_p$  are called via a *self* send in the methods  $m_1 \dots m_p$  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  $B_1 \dots B_k$ . This means that the intent is composed of the properties *abstractSelfDefinedLocally*, *concreteSelfDefinedInDescendant: B<sub>1</sub>, ...*, *concreteSelfDefinedInDescendant: B<sub>k</sub>*. Figure 4 illustrates this concept pattern graphically. The information expressed in this concept



**Figure 4.** *Concept pattern 3*

pattern identifies the *abstract interface* of a class, as well as the subclasses that provide a concrete implementation of this 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. In the *Magnitude* hierarchy, this concept pattern only occurs for the subhierarchies with root classes *Integer* and *ArithmeticValue*. For example, there is a concept with elements  $\{(ArithmeticValue, \{squared, *\}, \{=, -\})\}$  and properties  $\{abstractSelfDefinedLocally: ArithmeticValue, concreteSelfDefinedInDescendant: \{LargeInteger, Fraction,$

*Integer, SmallInteger, Float, FixedPoint, Point, Double* } }. In this example, the abstract methods {\*,-} in *ArithmeticValue* are called by other methods of the same class, but they are defined in descendant classes.

This concept pattern identifies the *hot spots* in an object-oriented application framework [JOH 88, DEM 98]. These hot spots are implemented by means of so-called *template methods* and *hook methods* [WIR 90, GAM 94]. 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.

## 5. Lessons Learned

Concerning the applicability of CA to understand software applications (in our specific case, class hierarchies) we identify three main issues about we have learned with this experience.

**Inference of unpredictable relationships:** This issue represents an advantage of using CA. The most important benefit is the unpredictability of the combination of the properties, generated by the algorithm. For example, the property *abstractSelfDefinedLocally* allows us to identify the abstract methods of a class, but if we group the properties *abstractSelfDefinedLocally* with *concreteSelfDefinedInDescendant* (concept pattern 3), this combination allows us to identify the cases of *template* and *hook methods*. This means, that one of the advantage of using CA is that the user must only specify (simple) properties to be applied on the elements, and the algorithm will generate automatically all the possible combinations of properties that are valid over the set of elements. The properties that are not valid for any element are discarded. Based on the combination, the developer is able to analyze the class hierarchy.

**Recognition of patterns:** In our specific application of CA to understand class hierarchies, we identify that the combination of properties in the concepts (seen from an abstract viewpoint) were repeated in the concept lattice. This fact helped us to identify behaviour patterns in the class hierarchies. We think that the classification of the results using abstractions clarifies the analysis and reduces the name of concepts to analyze. Thus, this is a further step for the analysis of concepts provided by CA, and this classification is not related to CA itself.

**Quality of Properties:** This issue is a crucial point and it is related to the second one. As we said previously, the useful properties are the result of an *iterative process* where we identify how generic (satisfied by most of the elements) or specific (satisfied by a small set of elements) the properties are. Depending on the chosen properties, concept analysis can produce too much concepts to analyze limiting the investigation about them. With too generic properties we will have a few concepts, which are not interesting enough because they will show a property satisfied by most of the elements (e.g., a concept that contains all the invocations that are called via self). With too specific properties we will have a large number of concepts, which are not interesting because they will show a property satisfied by only a small set of elements. In both, the result is a lattice with a mixture of non-useful and useful pieces of information,

and it makes difficult to get an optimal analysis of the result.

## 6. Related work

The mentioned related research are the only experiences that concretely used concept analysis to understand and support refactoring of class hierarchies. Godin and Mili [GOD 93, GOD 98] used concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. They showed how Cook's [COO 92] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. They suggest how the design of a class hierarchy implementing the detected interfaces could be organized in a way that optimizes the distribution of the methods over the hierarchy. In C++ and Java, Snelting and Tip [SNE 98] 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. As a result, they propose a new class hierarchy that is behaviorally equivalent to the original one. Similarly, Huchard[HUC 00]and Leblanc [LEB 00] applied concept analysis to improve the generalization/specialization of classes in a hierarchy. Based on this approach, Roume [ROU 02] defined metrics to measure the impact of refactorings to the class hierarchies.

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 superclass is essential to analyse and understand the kind of reuse that is achieved.

## 7. Conclusion and Future Work

In this paper we have summarized the analysis of class hierarchies using Concept Analysis. Based on information about self sends, super sends and invoked methods, we calculated the concepts using the *self* and *super* sends invocations as elements. 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. The approach allows us to identify the following features: document the subclass interface of a class; provide guidelines on how a class hierarchy 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 classes. Based on the diversity of features identified by the algorithm, 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 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. For example, Java has interfaces and C++ has multiple inheritance, but it is still unclear what the effect of this is on the structure of object-oriented hierarchies or on how reuse is achieved. 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. Another research avenue concerns the need to filter the information provided by the algorithm. In this experiment, only 80 concepts out of 125 (i.e., 64%) had a meaningful interpretation for us.

#### Acknowledgements

I 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). I would like to thank Stéphane Ducasse, Oscar Nierstrasz and Juan Carlos Cruz for reviewing drafts of the paper. I would like also to thank the anonymous reviewers for their helpful comments.

#### 8. References

- [AR02] ARÉVALO G., MENS T., "Analysing Object-Oriented Application Frameworks Using Concept Analysis", *Proceedings of the Inheritance Workshop at ECOOP 2002*, Andrew Black and Erik Ernst and Peter Grogono and Markku Sakkinen editors, University of Jyväskylä, 2002, p. 3–9.
- [BAR 70] BARBUT M., MONJARDET B., *Ordre et Classification*, Hachette, 1970.
- [CAS 95] CASAIS E., "Managing Class Evolution in Object Oriented Systems", NIERSTRASZ O., TSICHRITZIS D., Eds., *Object-Oriented Software Composition*, chapter 8, p. 201–244, Prentice Hall, 1995.
- [COO 92] COOK W. R., "Interfaces and Specifications for the Smalltalk-80 Collection Classes", *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications*, vol. 27(10) of *ACM SIGPLAN Notices*, ACM Press, October 1992, p. 1–15.
- [DEM 98] DEMEYER S., "Analysis of Overridden Methods to Infer Hot Spots", DEMEYER S., BOSCH J., Eds., *ECOOP '98 Workshop Reader*, vol. 1543 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998.
- [DUC 00] DUCASSE S., LANZA M., TICHELAAR S., "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems", *Proc. 2nd Int'l Symp. Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

- [FOW 99] FOWLER M., *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.
- [GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*, Addison-Wesley, 1994.
- [GAN 99] GANTER B., WILLE R., *Formal Concept Analysis: Mathematical Foundations*, Springer Verlag, 1999.
- [GOD 93] GODIN R., MILI H., “Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices”, *Proc. Int'l Conf. Object-Oriented Programs, Systems, Languages and Applications*, vol. 28 of *ACM SIGPLAN Notices*, ACM Press, October 1993, p. 394–410.
- [GOD 98] GODIN R., MILI H., MINEAU G. W., MISSAOUI R., ARFI A., CHAU T.-T., “Design of class hierarchies based on concept (Galois) lattices”, *Theory and Application of Object Systems*, vol. 4, num. 2, 1998, p. 117–134.
- [HUC 00] HUCHARD M., DICKY H., LEBLANC H., “Galois lattice as a framework to specify algorithms building class hierarchies”, *Theoretical Informatics and Applications*, vol. 34, 2000.
- [JOH 88] JOHNSON R. E., FOOTE B., “Designing Reusable Classes”, *J. Object-Oriented Programming*, vol. 1, num. 2, 1988, p. 22–35.
- [KUZ 01] KUZNETSOV S., OBĚDKOV S., “Comparing Performance of Algorithms for Generating Concept Lattices”, *Proc. Int. Workshop on Concept Lattices-based KDD*, 2001.
- [LEB 00] LEBLANC H., “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.
- [LIN 00] LINDIG C., “Fast Concept Analysis”, 2000.
- [LUC 97] LUCAS C., STEYAERT P., MENS K., “Managing Software Evolution through Reuse Contracts”, *Proc. 1st Euromicro Conf. Software Maintenance and reengineering*, IEEE Computer Society Press, 1997.
- [MIK 98] MIKHAILOV L., SEKERINSKI E., “The Fragile Base Class Problem and Its Solution”, *Proc. Int'l Conf ECOOP '98*, vol. 1445 of *Lecture Notes in Computer Science*, Springer Verlag, 1998, p. 355-382.
- [ROU 02] ROUME C., “Évaluation structurelle de la factorisation et la généralisation au sein des hiérarchies de classes: Introduction de métriques”, *L'Objet*, vol. 8, num. 1-2, 2002, p. 151-166.
- [SNE 98] SNELTING G., TIP F., “Reengineering Class Hierarchies Using Concept Analysis”, *ACM Trans. Programming Languages and Systems*, 1998.
- [WIL 81] WILLE R., “Restructuring lattice theory: An approach based on hierarchies of concepts”, *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, , 1981, p. 445-470.
- [WIR 90] WIRFS-BROCK R., WILKERSON B., WIENER L., *Designing Object-Oriented Software*, Prentice Hall, 1990.