Analysing Object Oriented Framework Reuse using Concept Analysis *

Gabriela Arévalo
Software Composition Group
University of Berne
Bern, Switzerland

arevalo@iam.unibe.ch

Tom Mens
Programming Technology Lab
Vrije Universiteit Brussel
Brussels, Belgium

tommens@vub.ac.be

ABSTRACT

This paper proposes the use of the formal technique of *Concept Analysis* to analyse how classes in an object-oriented inheritance hierarchy are coupled by means of the *inheritance* and *interfaces* relationships. To perform our analysis, we use the information provided by the *self-send* and *super-send* behaviour of each class in the hierarchy. Especially for large and complex inheritance hierarchies, we believe that this analysis can help in understanding the software, in particular with how reuse is achieved. Additionally, 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 position paper reports on an initial experiment with the *Magnitude* hierarchy in the *Smalltalk* programming language.

Keywords

concept analysis, inheritance hierarchy, interface, reuse

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. Defining a class in an application 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 between the classes in the inheritance hierarchy.

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 *properties* (referred to as *attributes* in CA literature) [1]. These groupings are called *con-*

---

*In ECOOP 2002: Proceedings of the Inheritance Workshop, A.Black, E.Ernst, P.Grogono and M. Sakkinen (Eds.), pp. 3-9, University of Jyväskylä, 2002

[1] We prefer to use the terms *element* and *property* instead of the

cepts 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 [11, 4].

The advantage of CA is that this technique allows us to infer commonalities between elements based only on the specification of simple properties satisfied by each element. Since the user only needs to specify the properties of interest on each element, he does not need to think about all possible combination of these properties, since these groupings are made automatically by the CA algorithm.

In this paper, we report on an experiment that uses concept analysis to analyse an existing inheritance hierarchy with the aim to better understand whether and how inheritance is used in practice to achieve reuse.

2. APPLYING CONCEPT ANALYSIS TO INHERITANCE HIERARCHIES

We will use the CA technique to 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 exported by the classes. 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*.

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

Classification of the sender
- *(Class, selector)* satisfies the predicate
  calledViaSelf
  if *selector* is called via a self send in *Class*

---

terms *object* and *attribute* in this paper because the terms *object* and *attribute* have a very specific meaning in the object-oriented programming paradigm.

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

Classification of the implementation

- *(Class, selector)* satisfies the predicate
  hasConcreteImplementationIn: *otherClass*
  if *selector* is implemented as a concrete method in *otherClass*

- *(Class, selector)* satisfies the predicate
  hasAbstractImplementationIn: *otherClass*
  if *selector* is implemented as an abstract method in *otherClass*

Classification of the relationship between sender and implementor classes

- *(Class, selector)* satisfies the predicate
  isImplementedInAncestor: *ancestorClass*
  if *ancestorClass* is an ancestor class (i.e., a direct or indirect superclass) of *Class* that implements *selector*.

- *(Class, selector)* satisfies the predicate
  isImplementedInDescendant: *descendantClass*
  if *descendantClass* is a descendant class (i.e., a direct or indirect subclass) of *Class* that implements *selector*.

- *(Class, selector)* satisfies the predicate
  isImplementedLocally
  if *Class* implements *selector*. This means that there is an implementation of *selector* in the same class that calls it.

CA properties are then defined as conjunctions obtained by taking one predicate from each subgroup. For example,

- Property **concreteSuperCaptureIn:** *Class*
  is a conjunction of the three predicates *calledViaSuper* and *hasConcreteImplementationIn: Class* and *isImplementedInAncestor: Class*.

- Property **concreteSelfCaptureLocally:** *Class*
  is a conjunction of the three predicates *calledViaSelf* and *hasConcreteImplementationIn: Class* and *isImplementedLocally*.

- Property **concreteSelfCaptureInAncestor:** *Class*
  is a conjunction of the three predicates *calledViaSelf* and *hasConcreteImplementationIn: Class* and *isImplementedInAncestor*.

- Property **concreteSelfCaptureInDescendant:** *Class*
  is a conjunction of the three predicates *calledViaSelf* and *hasConcreteImplementationIn: Class* and *isImplementedInDescendant: Class*.

- Property **abstractSelfCaptureLocally:** *Class*
  is a conjunction of the three predicates *calledViaSelf* and *hasAbstractImplementationIn: Class* and *isImplementedLocally*.

These are only some of the possible properties resulting from a conjunction of the predicates presented previously. We only specify those properties which will be used for the case study.

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 1. Using the information present in this table, we can run the the CA algorithm to compute *concepts*. From the abstract example given in Table 1, the CA algorithm will automatically compute the following concepts (among others):

- **Concept 1** has singleton element set { (C_4, s_2) } and property set { *concreteSelfCaptureLocally:* C_4, *concreteSuperCaptureIn:* C_5 }
  This concept means that C_4 has a *self* and *super* send of the selector s_2, and this selector is implemented as a *concrete* method in the same class C_4, and in one ancestor C_5.

- **Concept 2** has element set { (C_4, s_2), (C_4, s_5), (C_4, s_6), (C_4, s_7), (C_4, s_8) } and singleton property set { *concreteSelfCaptureLocally:* C_4 }
  This concept means that only the selectors s_2, s_5, s_6, s_7 and s_8 are called via a *self* send that is captured by *concrete* method implementations in the class C_4.

- **Concept 3** has element set { (C_1, s_1), (C_2, s_2), (C_3, s_3), (C_3, s_1), (C_4, s_4), (C_4, s_2) } and singleton property set { *concreteSuperCaptureIn:* C_5 }
  This concept 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.

Based on the information given by the concepts, we could understand how the classes are related using different criteria. In our case, we analyzed two specific features of a class hierarchy: *inheritance* and *interface*.

3. CASE STUDY
3.1 Terminology
We will first introduce some terminology (similar to the one used in [8]) that will be needed for the rest of the paper:

- The *client interface* of a class is the set of all selectors that are implemented in the class and to which direct sends can be made[8]. In Java, the client interface is the set of all `public` methods. In Smalltalk, the client interface is the set of all methods (since there is no visibility mechanism in Smalltalk).

- The *subclass interface* of a class is the set of all selectors that are implemented in the class and to which self sends can be made by subclasses. In Java, the subclass interface is the set of all `public` and `protected` methods. In Smalltalk, the subclass interface is the set of all methods.

- The *overriding interface* of a class is the set of all selectors that are implemented in the class and to which super sends can be made by methods implemented in descendants. In Java, the overriding interface is a subset of the subclass interface, namely all those `public` or `protected` methods that are not `final`, since `final` methods cannot be overridden in subclasses. In Smalltalk, every method defined in a class can be overridden.

	concreteSelfCaptureLocally: C_4	concreteSuperCaptureIn: C_5
(C_1, s_1)	False	True
(C_2, s_2)	False	True
(C_3, s_3)	False	True
(C_3, s_1)	False	True
(C_4, s_4)	False	True
(C_4, s_2)	True	True
(C_4, s_5)	True	False
(C_4, s_6)	True	False
(C_4, s_7)	True	False
(C_4, s_8)	True	False

Table 1: Elements and their satisfied properties in an inheritance hierarchy

- The *abstract interface* of a class is the set of all selectors that are defined abstract in the class and are required to be implemented by a concrete method in descendants. In Java, the abstract interface is the set of all `abstract` methods, which is a subset of the subclass interface. In Smalltalk, the abstract interface is a subset of the overriding interface, since abstract methods are modelled by `self subclassResponsibility` and have to be overridden in subclasses.

- The *internal interface* of a class is 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.

- The *actual client/subclass/overriding/abstract interface* of a class is the set of all selectors that are implemented by a class and that are actually accessed (via an invocation, self send or super send, respectively) in the implementation. Obviously the actual interface of a class is always a subset of the total client/subclass/overriding/abstract interface of the class.

3.2 Experimental setup

The abstract example explained in 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 (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. Future experiments will be carried out on other well-known Smalltalk hierarchies such as *Collection*, *Model*, *View* and *Controller*.

Based on results provided by the CA algorithm, we analyzed the relationships between the classes in terms of *inheritance* and *interface*. Compared with the example presented previously, we worked with 248 elements and 73 properties, and the algorithm gave 125 concepts as a result. The elements and properties have been extracted with SOUL, a logic meta-programming language built on top of –and tightly integrated with– Smalltalk [13]. The CA analysis tool itself, that computes the concept lattice, was implemented directly in Smalltalk, and uses the results of the SOUL predicates as input.

The information about the inheritance hierarchy is extracted from the resulting concepts of the CA algorithm. As we said previously, the concepts associate sets of elements with sets of properties. The properties will be a subset of { *concreteSuperCaptureIn: Class, concreteSelfCaptureInDescendant: Class, abstractSelfCaptureLocally: Class, concreteSelfCaptureLocally: Class, concreteSelfCaptureInAncestor: Class, ...* }. If we abstract the argument *Class* out of these properties, we find that many concepts contain the same set of properties. This commonality between concepts allows us identify *patterns of concepts*.

3.3 Concept patterns

A *concept pattern* consists of a description, examples related to the *Magnitude* class hierarchy, a figure that illustrates the pattern and an analysis about possible implications of the pattern with respect to software understanding and reuse. [3]

Concept Pattern 1: Self sends captured locally

Description. A set of selectors $m_1,..., m_p$ are called via a *self* send in a class *B* and they are implemented in the same class. Figure 1 shows this concept pattern graphically. It occurs in 21 concepts of the *Magnitude* concept lattice. For example, **Concept 61** has elements {*(AssociationTreeWithParent, {expanded:, depth, parent})*} and singleton property set {*concreteSelfCaptureLocally: AssociationTreeWithParent*}
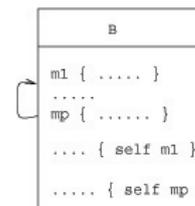
Figure 1: Concept pattern 1

Analysis. This concept pattern is useful to document the *internal interface* of a class, which captures class-specific behaviour. The number of elements present in the concept can also be used to give

---

[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]In the examples section of each concept pattern we use numbers for the concepts. This number is only an id automatically associated to the concept by the algorithm. To make the notation for concepts more compact, we will group together all selectors belonging to the same class. For example if we have in a concept with the elements {(C_1, s_1), (C_1, s_2), (C_1, s_3), (C_2, s_4)}, we will show it as {$(C_1, \{s_1, s_2, s_3\})$,(C_2, s_4)}

an indication of to which extent the class is reusing parent methods, by comparing the number of locally captured self references with the number of methods inherited from the parent.

Concept pattern 2: Self sends captured in ancestor

Description. A set of selectors $m_1,..., m_p$ are called via *self* send in the classes $A_1,..,A_n$ and the selectors are implemented in the class B, which is a common superclass of $A_1,..,A_n$. Figure 2 displays this concept pattern in a general way. It occurs in 9 concepts of the *Magnitude* concept lattice.
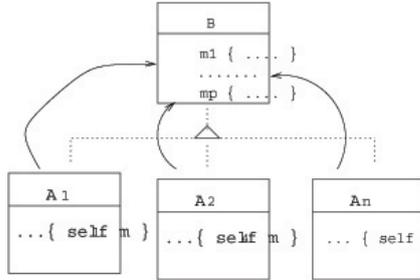

Figure 2: Concept pattern 2

Analysis: In general terms, this concept pattern is useful to detect the *actual subclass interface* of a class. This pattern is also useful to find out whether the same set of selectors called in a set of classes are implemented in a specific superclass, which does not have to be the same for each class-selector pair. This can be more useful for finding a common interface for the subclasses.

Considering the direct/indirect inheritance relationship between B and A_i and among the different A_is, we can distinguish 2 different cases:

Case 1. The selectors $m_1,...,m_p$ are called in a set of classes $A_1,..,A_n$. The selectors are implemented in a common direct superclass B of all the classes. The classes $A_1,..,A_n$ are sibling classes, i.e., they have no inheritance dependencies between them. Figure 3 shows this case. For example, **Concept 73** has elements {*(LargePositive Integer, {digitLength, digitAt:}), (LargeNegativeInteger, { digitLength, digitAt:})}* and singleton property set {*concreteSelfCaptureInAncestor: LargeInteger*}
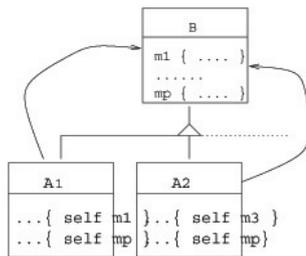

Figure 3: Concept pattern 2 - Case 1

Analysis of case 1. This case can also be used to identify common code in sibling classes that is useful to refactor in the common superclass. For example, concept 73 implies that, to a certain extent, sibling classes *LargePositiveInteger* and *LargeNegativeInteger* reuse the behaviour defined in their superclass *LargeInteger* in the same way. An investigation of the actual source code to find out which method performs the self sends *digitLength* and *digitAt:* learns us that both self sends are invoked from within the implementation of the method *compressed* in both sibling classes. Moreover, 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*.

Case 2. The selectors $m_1,...,m_p$ are called in a set of classes $A_1,..,A_n$. The selectors are implemented in a common ancestor B of all the classes. The classes $A_1,..,A_n$ may have inheritance dependencies between them. Figure 4 shows this case. For example, **Concept 78** has elements {*(LimitedPrecisionReal, {floorLog:, truncated:}), (Float, {quo:, rem:}), (Double, quo:)}* and singleton property set {*concreteSelfCaptureInAncestor: Number*}. *LimitedPrecisionReal* is a common superclass of *Float* and *Double*. They are all subclasses of *Number*.
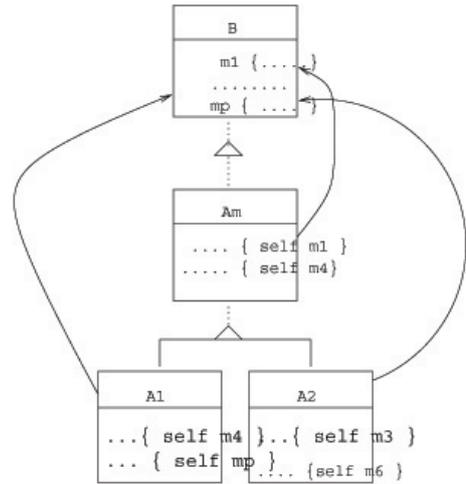
Figure 4: Concept pattern 2 - Case 2

Analysis of case 2. In this specific case, concept 78 captures the *actual subclass interface* of *Number*, which contains all methods in *Number* that are reused via self sends by at least on of its descendants. This is very important information, since in Smalltalk all methods are public, so it is very difficult to know in practice which methods are actually being reused in subclasses. In Java, the distinction between public, protected and private methods partly solves this problem, but we still don't know which of the public and protected methods are actually reused by subclasses. In principle, every method in the protected interface should also be present in the actual subclass interface, since it is of not much practical use to declare a method protected if it is never called by subclasses.

Concept pattern 3: Super call

Description. A set of selectors $m_1,..., m_p$ are called via a *super* send in the classes $A_1,..,A_n$ and the selectors are implemented in the class B, which is a common ancestor of $A_1,..,A_n$. Figure 5 illustrates this concept pattern in a general way. It occurs in 8 concepts of the *Magnitude* concept lattice.

Analysis: In general terms, this concept pattern can be used to detect the *actual overriding interface* of a class. Considering the direct/indirect inheritance relationship between B and A_i and among the different A_is, we can distinguish 2 different cases:
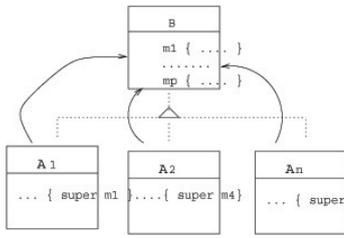
Figure 5: Concept Pattern 3

Case 1. The selectors $m_1,...,m_p$ are called in a set of classes $A_1,..,A_n$. The selectors are implemented in a common ancestor B of all the classes. The classes $A_1,..,A_n$ are sibling classes, i.e., they have no inheritance dependencies between them. Figure 6 illustrates this case. For example, **Concept 105** has elements $\{(Float, \{>, \geq, \leq\}), (Double, \{>, \geq, \leq\}), (SmallInteger, \{>, \geq, \leq\}), (LargeInteger, \{>, \geq, \leq\})\}$ and singleton property set $\{concreteSuperCaptureIn: Magnitude\}$
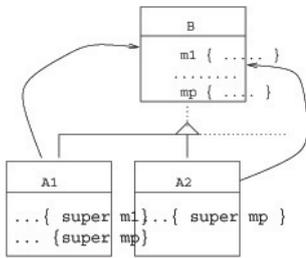
Figure 6: Concept pattern 3 - Case 1

**Analysis of case 1.** This concept pattern can be used to document and capture the *actual overriding interface* of a class. For example, the overriding interface of *Magnitude* is $\{>, \geq, \leq\}$ according to the information given by the Concept 105. The concept pattern can also be used to 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.

Case 2. The selectors $m_1,...,m_p$ are called in a set of classes $A_1,..,A_n$. The selectors are implemented in a common ancestor B of all the classes. The classes $A_1,..,A_n$ have some *direct* inheritance dependencies between them. Figure 7 illustrates this case. For example, **Concept 114** has elements $\{(LargePositiveInteger, computeGCD:), (LargeInteger, \{ \backslash\backslash, bitAnd:, bitXor:, bitOr:, bitShift:, lessFromInteger:, =, sumFromInteger:, equalFromInteger:, productFromInteger:\}), (SmallInteger, \{bitShift:, <, //, =\})\}$ and singleton property set $\{concreteSuperCaptureIn: Integer\}$. *LargePositiveInteger* is a subclass of *LargeInteger* and all of them are subclasses of *Integer*.

Analysis of case 2. This case can be used to document *implementation inheritance*[9]. This means that the methodname -where the selector is called- and the selector name are the same. In the case of Concept 114, we see that this behaviour delegation is not so direct because before the *super* send, there is a statement with a *primitive* action.

Concept pattern 4: Self send captured locally and in descendant
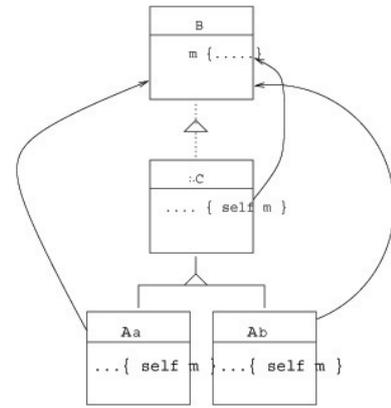
Figure 7: Concept pattern 3 - Case 2

Description. A set of selectors $m_1,..., m_p$ are called via a *self* send in the class A and the selectors are implemented in the classes A and $B_1,..,B_k$, which are subclasses of A. Figure 8 shows this concept pattern in a general way. It occurs in 31 concepts of the *Magnitude* concept lattice. For example, **Concept 69** has element set $\{(Number, \{raisedTo:, sqrt, ln, truncated\})\}$ and properties set $\{concreteSelfCaptureInDescendant: Float, concreteSelfCaptureInDescendant: Double, concreteSelfCaptureLocally: Number\}$.
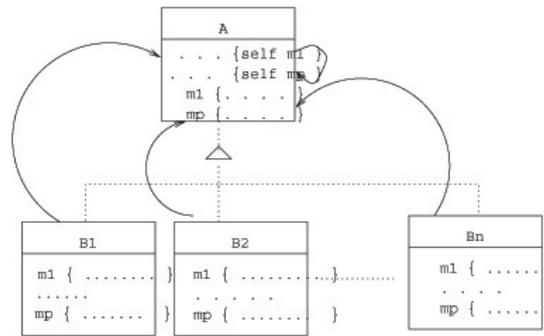
Figure 8: Concept pattern 4

Analysis. This concept pattern can be used to document which specific methods are overriden 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 with subclass-specific behaviour.

Concept pattern 5: Self send locally with super delegation

Description. A set of selectors $m_1,..., m_p$ 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 of A. Figure 9 illustrates this concept pattern in a general way. It occurs in 4 concepts of the *Magnitude* concept lattice. In this pattern, we do not distinguish between different cases because the 4 concepts fulfilling this pattern are similar except that in some cases the superclass is not direct ancestor. For example, **Concept 48** has elements $\{(SmallInteger, \{>, \geq, \leq\})\}$ and properties set $\{(concreteSuperCaptureInAncestor: Magnitude, concreteSelfCaptureLocally: SmallInteger\}$
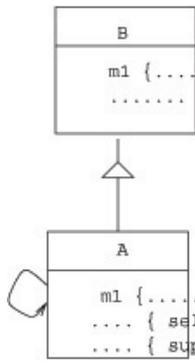
Figure 9: Concept pattern 5

Analysis. This concept pattern documents delegation between methods in the same class and with the superclass. Having a look at the code for the 4 concepts, in all the cases, the method that calls the selector s_1 via a *super send* has the same name as the selector (s_1). For example, in *SmallInteger* the selector $\geq$ is called via a *super send* in the method $\geq$. This means that if there is a specific action to be executed (when *self send* is called), this behaviour is defined in the superclass, because the message is delegated by the *super send* call. The set of properties allows us to see that this pattern is a combination of pattern 1 (where the predicate *concreteSelf-CaptureLocally:* is captured) and pattern 3 (where the predicate *concreteSelfCaptureInAncestor:* is captured).

Concept pattern 6: Template and hook methods

Description. 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 $B_1 \ldots B_k$ of A. Figure 10 illustrates this concept pattern in a general way. It occurs in 7 concepts of the *Magnitude* concept lattice.
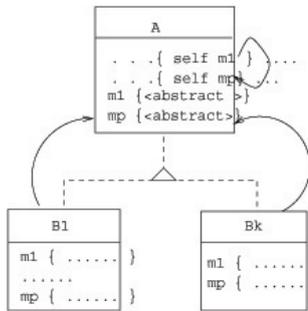
Figure 10: Concept pattern 6

Analysis. This concept pattern is essential in understanding an object-oriented application framework. More specifically, it allows us to identify the *hot spots* in the application framework [7, 2]. These hot spots are very often implemented by means of so-called *template methods* and *hook methods* [12, 3]. 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. This is precisely the information that is captured by concept pattern 6.

Seen in another way, 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 interface. This information is essential 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, concept pattern 6 only occurs for the subhierarchies with root classes *Integer* and *ArithmeticValue*.

We can distinguish 2 different subcases of this concept pattern, depending on how the classes A and $B_1 \ldots B_k$ are related by inheritance:

Case 1. The selectors $m_1 \ldots m_p$ are called with a *self* send in the class A and the selectors are implemented in direct subclasses $B_1 \ldots B_k$ of A. Figure 11 shows this concept pattern in a general way. For example, **Concept 56** has element set $\{(Integer, \{digitAt:put:, digitLength, digitAt:\})\}$ and properties set $\{abstractSelfCaptureLocally: Integer, concreteSelfCaptureInDescendant: SmallInteger, concreteSelfCaptureInDescendant: LargeInteger\}$
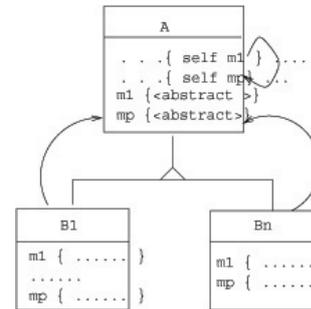
Figure 11: Concept pattern 6 - Case 1

Analysis of case 1. In this case, we can identify a potential problem in the hierarchy. The subclasses do not implement all the *abstract* methods of the superclass. This can result in a subtle error if we call a method in the superclass that calls an *abstract* method, especially in Smalltalk which does not make an explicit difference between concrete and abstract classes (we can create instances of both).

Case 2. The selectors $m_1 \ldots m_p$ are called with a *self* send in the class A and the selectors are implemented in the classes $B_1 \ldots B_k$, which are indirect descendants of A. The descendants have inheritance dependencies between them. Figure 12 shows this concept pattern in a general way. For example, **Concept 31** has element set $\{(ArithmeticValue, \{*, -\})\}$ and properties set $\{abstractSelfCaptureLocally: ArithmeticValue, concreteSelfCaptureInDescendant: LargeInteger, concreteSelfCaptureInDescendant: Fraction, concreteSelfCaptureInDescendant: Integer, concreteSelfCaptureInDescendant: SmallInteger, concreteSelfCaptureInDescendant: Float, concreteSelfCaptureInDescendant: FixedPoint, concreteSelfCaptureInDescendant: Point, concreteSelfCaptureInDescendant: Double\}$

Analysis of case 2. This case is a bit tricky, because the results can be considered as a combination of two different situations: the concrete implementation of abstract methods in descendants of a class, and the overriding of these concrete implementations in further descendants. In concept 31, the relationship between classes *ArithmeticValue* and *Integer* is that the abstract methods $\{*,-\}$ in *ArithmeticValue* are implemented by concrete methods in subclass *Integer*. Moreover, these concrete methods are overridden by further descendants *SmallInteger* and *LargeInteger* that optimise the
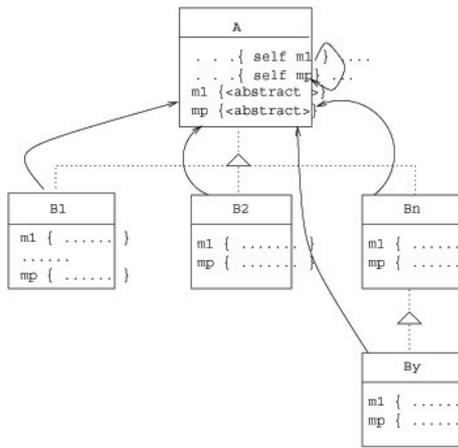
Figure 12: Concept pattern 6 - Case 2

implementation by making use of Smalltalk primitives.

4. RELATED WORK

Godin and Mili [5, 6] used concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. They showed how Cook's [1] 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 [10] analysed a class hierarcy 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.

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

In this position paper we proposed to analyse inheritance hierarchies using Concept Analysis. We took into account two main aspects among the classes: inheritance and interface relationships. Based on information about abstract methods, self sends and super sends, we calculated the concept lattice for a well-known inheritance hierarchy: the Smalltalk *Magnitude* hierarchy. Then, we analysed the results after classifying the generated concepts into concept patterns. Each concept pattern allowed us to discover a number of interesting non-documented relationships (based on self sends and super sends) among classes in a hierarchy. Especially for large inheritance hierarchies, this information is crucial for understanding the software.

Based on the preliminary results of our experiments, we believe that Concept Analysis is a promising technique in software understanding and re-engineering. Obviously, it is only a first step. One possible avenue of research is the refinement of the technique and the focus on more specific relationships and more behavioural information to get more specific results. We think that the technique can help us: (1) to provide guidelines on how a given object-oriented

application framework should be customised; (2) to identify and document hot spots in an object-oriented application framework; (3) to automatically extract and document implicit coding conventions used while building an inheritance hierarchy; (4) to detect and document the type of inheritance used in (parts of) an inheritance hierarchy.

6. REFERENCES

[1] W. R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15. ACM Press, October 1992.

[2] 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.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.

[4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[5] 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 (OOPSLA93)*, volume 28 of *ACM SIGPLAN Notices*, pages 394–410. ACM Press, October 1993.

[6] 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.

[7] R. E. Johnson and B. Foote. Designing reusable classes. *J. Object-Oriented Programming*, 1(2):22–35, Feb. 1988.

[8] J. Lamping. Typing the specialization interface. In *Proc. Int'l Conf. Oject-Oriented Programs, Systems, Languages and Applications (OOPSLA93)*, volume 28 of *ACM SIGPLAN Notices*, pages 201–214. ACM Press, October 1993.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[10] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1998.

[11] 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.

[12] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[13] 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.