Identifying Traits with Formal Concept Analysis

Adrian Lienhard
Software Composition Group
University of Bern
(Switzerland)
lienhard@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
University of Bern
(Switzerland)
and
Language and Software
Evolution Group
Université de Savoie (France)
ducasse@iam.unibe.ch

Gabriela Arévalo
Software Composition Group
University of Bern
(Switzerland)
arevalo@iam.unibe.ch

ABSTRACT

Traits are basically mixins or interfaces but with method bodies. In languages that support traits, classes are composed out of traits. There are two main advantages with traits. Firstly, decomposing existing classes into traits from which they can be recomposed improves the factoring of hierarchies. Secondly it increases the library reuse potential by providing more reusable traits. Identifying traits and decomposing class hierarchies into traits is therefore an important and challenging task to facilitate maintainability and evolution. In this paper we present how we use Formal Concept Analysis to identify traits in inheritance hierarchies. Our approach is two-staged: first we identify within a hierarchy *maximal groups of methods* that have a set of classes in common, second we cluster *cohesive groups of methods* based on method invocations as potential traits. We applied our approach on two significant hierarchies and compare our results with the manual refactorization of the same code which was done by the authors of traits.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Languages, Design

Keywords

Traits, Mixins, Formal Concept Analysis, Logical Views

1. INTRODUCTION

A trait is essentially a set of methods, similar to mixins or interfaces but with method bodies, featuring a reuse mechanism complementary to inheritance [6, 20]. The advantage of traits for supporting better composition and reuse has been recognized by languages such as Perl 6, Squeak [17], Scala [19], Slate and Fortress [11]. These languages have been extended to support the traits mechanism. Several works are currently performed to introduce traits at the level of type systems [10, 21]. The introduction of a new construct like traits into a programming language, which provides support for writing better software with increased potential for reuse, raises the issue of reengineering existing code to exploit this construct. Clearly there is a need for a technique that supports the identification of potential traits in existing libraries and applications, so that they can be easily refactored to exploit advantages of traits. Up until now, the identification of traits has been a manual task [4].

In this paper we describe an approach to trait identification based on Formal Concept Analysis. We apply our approach to two major Smalltalk class hierarchies. We validate our results by comparing them with the results obtained by manual refactorization of the same code base [4].

The problems that the reengineer is facing are manifold:

- A concrete class inherits its behavior from several ancestors masking each others. Therefore concrete interfaces are scattered over several classes with possible cancellation and redefinition [1].
- Methods' late-binding leads to yoyo effects and make the code difficult to follow and understand [24, 26, 9].
- Identifying cohesive groups of methods that can be parametrized is challenging since several degrees of parametrization can be achieved and methods are spread over multiple classes in a hierarchy.

Our approach is based on Formal Concept Analysis (FCA)[1] to help in the identification of groups of methods in existing hierarchies and in single classes and traits that could be turned into a (sub)trait. FCA is an appropriate technique to cope with discovering new kinds of "patterns", as in our case set of methods, without knowing in advance how they are composed.

Our approach is two-staged: (i) we restructure the hierarchy and identify traits by detecting *maximal groups of methods shared by a set of classes* within a hierarchy, (ii) we cluster *cohesive groups of methods* based on method invocations as potential traits. The

[1]FCA [14] is a branch of lattice theory that allows us to identify meaningful groupings of "objects" that have common "attributes". To avoid unfortunate clash with object-oriented terminology, we use the terms *elements* and *properties*. For details see Appendix of this paper.

ASE'05, November 7–11, 2005, Long Beach, California, USA.

tool implementing it proposes behavior preserving refactorings but does not transform hierarchies automatically. It presents a solution which serves as a good starting point to refactor a system and which supports program understanding. For the refactoring we are able to retrieve relevant information, *i.e.*, potential traits, classes and the structure of the new inheritance hierarchy.

The contributions of the paper are:

- Analysis of coding idioms that shows the symptoms of the lack of traits applications in existing single inheritance class hierarchies.
- Use of FCA to identify trait refactoring opportunities based on a two-staged approach: using method commonality within a hierarchy and invocation relationships between methods.
- First evaluation of our approach compared with the results obtained by manual refactorings of significant hierarchies presented by the authors of traits [4].

The outline of the paper is the following: first we analyze the code symptoms that can reveal the need to apply traits in existing hierarchies (Section 2), then we present traits in a nutshell (Section 3). Section 4 presents our approach of using FCA to identify traits. In Section 5 we discuss the results we obtained on the Stream and Collection Hierarchies of Squeak, an open-source Smalltalk, and compare with the manual refactoring of the same code. We discuss strengths and weaknesses of our approach before concluding.

2. REUSE IDIOMS IN SINGLE INHERITANCE HIERARCHIES

In class-based object-oriented programming languages without multiple inheritance or traits, the programmers are forced to duplicate code [12] or implement methods too high in the hierarchy. This section discusses three known common idioms in Smalltalk libraries used to work around the restrictions of single inheritance: method *duplication*, methods implemented too high in the hierarchy and then *cancelled* in subclasses, and not implemented methods.

Duplicated methods. When reusing code between two classes that are not in the same class hierarchy, methods are duplicated between different classes in the system. Figure 1 shows a simple case of duplication (used in several subsequent sections of this paper). It shows a part of the Smalltalk-80 Stream Hierarchy with the root class Stream, its subclass PositionableStream (both classes are merged in the figure for simplicity) and the three concrete stream classes WriteStream, ReadWriteStream and ReadStream. The method next, used for reading from the stream, is duplicated in the classes ReadWriteStream and ReadStream. If we push it up to the common superclass, WriteStream would inherit this behavior although it is not appropriate.

Methods implemented too high in the hierarchy. This is a common, and in most cases probably the best solution to share code when it is applicable. The idiom is to implement the methods in the common superclass of the classes that should share them. In the other subclasses the unnecessary inherited methods are *cancelled* by overriding the original implementation with a method that raises an exception at run-time.

A commonly used variation is the combination with the Template Method design pattern [13]. The idea is to implement the template methods that should be shared too high in the hierarchy and

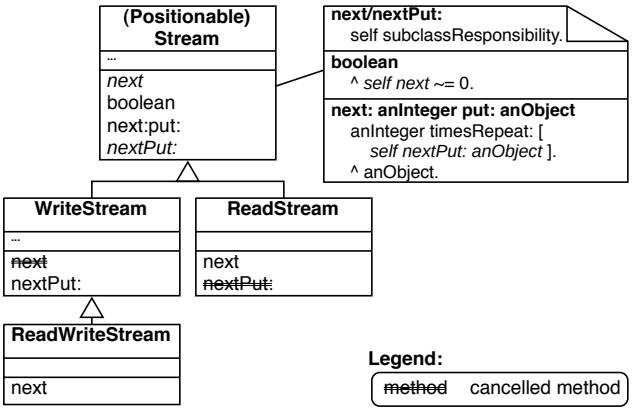

Figure 1: A part of the Smalltalk Stream Hierarchy where methods are duplicated or defined too high.

only cancel (or mark as subclass-responsibility [2]) the hook method. This has the effect that only the hook method needs to be cancelled in the inappropriate classes. The other methods do not have to be cancelled explicitly because they indirectly call the hook method which is cancelled.

Figure 1 shows this idiom as follows. The template method next:put: writes a number of times an object on the stream. It is implemented in Stream to be shared among the different write streams. There exist eight other classes apart from WriteStream and ReadWriteStream that use this and related methods. The method calls nextPut: which is declared as *abstract* in Stream. This hook method is now, according to the kind of write stream, implemented in each of the concrete classes and it is explicitly cancelled in the others (like ReadStream). Methods like next:put: do not have to be modified in any of the subclasses of Stream.

The method boolean, which implements reading a boolean value from a binary stream is defined similarly to next:put:. It has the hook method next which is explicitly cancelled in WriteStream and implemented in ReadWriteStream and ReadStream. In this case, though, the concrete implementations of next are identical as discussed in the previous paragraph about method duplication.

Not implemented methods. Another symptom of missing reuse possibility are methods that would ideally be understood by a class but are not implemented[3] or inherited, often because they would cause further code duplication or because they were not needed at the time the class was written. A prominent example is the iteration protocol in the Smalltalk Collection library. Often only a fraction of the well-known methods are implemented. Commonly the implementation of the collection protocol, *e.g.*, iterating over the collection, is implemented by delegating to a collection which is hold as an instance variable.

For example the class Path in Squeak implements an ordered sequence of points and it inherits from DisplayObject which implements display primitives to present information on the screen. Path has an instance variable collectionOfPoints and it implements well known methods of the collection protocol [3] such as at:, at:put:, collect:, and select:. But those methods are only a fraction of the

[2]In Smalltalk, marking a method as abstract is possible anywhere in the hierarchy and is done by sending the message subclassResponsibility which raises a signal at run-time.

[3]Smalltalk is a dynamically typed language, therefore a method can call another one that does not even exist.

protocol that is often used. For example the method do: to iterate over the collection of points or the counterpart of select:, reject:, are not implemented.

3. TRAITS IN A NUTSHELL

Traits are an extension of single inheritance with a similar purpose as mixins but avoid their problems [20]. Traits are groups of methods that serve as building blocks for classes. They are primitive code reuse units. Traits allow common behavior to be factored out to form an intermediate level of abstraction between single methods and complete classes. A trait consists of *provided methods* that implement its behavior, and of *required methods* that parameterize the provided behavior. Traits cannot specify any state, and the methods provided by traits never directly access it. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class or composite trait. These glue methods connect the traits together and serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.
- *Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises when we combine two or more traits that provide identically named methods that do not originate from the same trait. Conflict resolution is explicit and based on the implementation of a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows the exclusion of the conflicting method from all but one trait. In addition traits allow *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden. Traits can be composed from subtraits. The composition semantics is the same as explained above with the only difference being that the composite trait plays the role of the class. Whereas inheritance is used to derive one class from another, traits are used to achieve structure and reusability *within* a class definition.

Example: Geometric Objects. Suppose that we want to represent a graphical object such as a circle or square that is drawn on a canvas. Such a graphical object can be decomposed into three reusable aspects — its geometry, its color and the way that it is drawn on a canvas.

Figure 2 shows this for the class Circle. First of all, the geometry of a circle is specified with a trait TCircle. Furthermore the color is expressed using a trait TColor, and the behavior for drawing an object on a canvas is provided by a trait TDrawing:

- TCircle defines the geometry of a circle: it requires the methods center, center:, radius, and radius: and provides methods such as bounds, hash, and =.
- TDrawing requires the methods drawOn: and bounds and provides the methods draw, refresh, and refreshOn:.

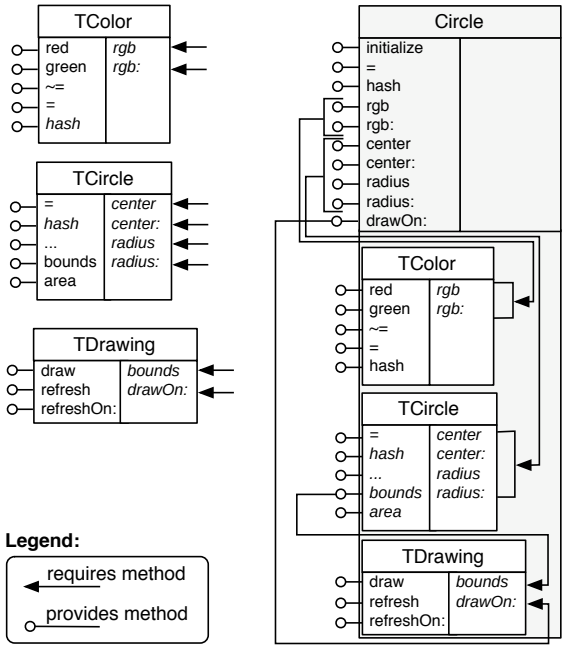

Figure 2: Left: Three traits TColor, TCircle, and TDrawing. Right: The class Circle composed from these traits. The class Circle resolves conflicts by redefining hash and =.

- TColor requires the methods rgb, rgb: and provides all kind of methods manipulating colors. We only show the methods hash and = as they will be conflicting with others at composition time.

The class Circle is then defined as follows: it specifies three instance variables center, radius, and rgb and their respective accessor methods. It is composed from the three traits TDrawing, TCircle, and TColor. As there is a conflict for the methods hash and = between the traits TCircle and TColor, we alias those methods in both traits to be able to access them in the methods hash and = of the class Circle resolving the conflicts.

4. APPLYING FCA TO IDENTIFY TRAITS AND RESTRUCTURE HIERARCHIES

Traits improve the sharing of code between classes and increase the reuse potential. In a single inheritance hierarchy with traits, the commonly used coding idioms identified in Section 2 can be avoided.

Our approach restructures inheritance hierarchies and introduces traits to solve the identified problematic idioms. We detect common behavior which is shared among classes by using traits, and then refine the obtained classes and traits by identifying traits within them. The result is an optimally factored class hierarchy with fine-grained traits and with the identical behavior like the original hierarchy.

Our approach applies Formal Concept Analysis (FCA) [14] which is a branch of lattice theory that allows us to identify meaningful groupings of *elements* that have common *properties*. The groups are named *concepts*. Concepts can be ordered in a lattice according to a partial order. For more details of FCA, we refer the reader to the Appendix of this paper.

Overview of our approach. Our approach is divided into two *stages*. The first stage restructures the hierarchy and identifies traits; the second stage then identifies traits within the obtained classes and traits of stage one. The principle of stage one is to detect the set of methods that a class should implement, inherit or obtain from a trait. This information then serves as input to FCA from which we retrieve *maximal groups of methods that a set of classes have in common*. From these groups of methods we can then identify traits and reconstruct the class hierarchy. In stage two we apply FCA on individual classes and traits to identify cohesive groups of methods by analyzing invocations.

Outline of the process. The two stages of the process are organized in six steps shown in Figure 3. The steps 1 – 4 form the *first stage* of the process in which the hierarchy is restructured and traits are identified using FCA. In the *second stage*, performed in step 5 and 6, we once again apply FCA to identify traits, but this time *within* classes and traits which were obtained from the first stage.

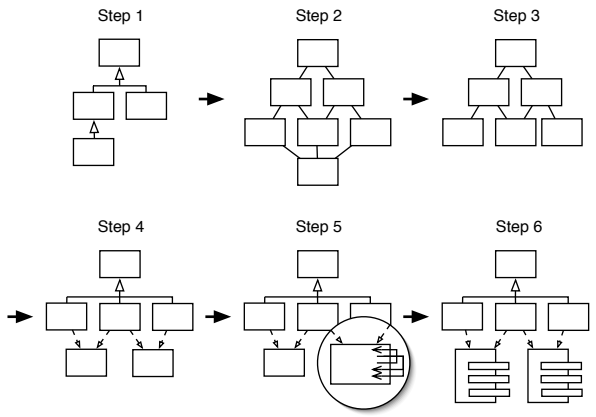

Figure 3: Process of restructuring a hierarchy and identifying traits.

The 6 steps, which are discussed in detail in the subsequent Sections 4.1 - 4.6, perform the following actions: step 1 generates the input for FCA by analyzing the classes that are selected by the reengineer. Step 2 produces the FCA concept lattice. Step 3 processes the lattice to reduce information and remove not interesting concepts. Step 4 identifies traits and infers the new hierarchy from the lattice of the previous step. Step 5 and 6 are applied to each of the obtained classes and traits from Step 4. Step 5 again applies FCA to decompose a class or a trait into (sub)traits. The lattice produced in Step 5 proposes potential method groups to the reengineer in Step 6. Step 4 and 6 are partly manual tasks which are supported by the tool.

4.1 Generating the Input to FCA

For each class selected by the reengineer all concrete methods that are implemented locally or that are inherited are collected. Methods that cancel behavior or are abstract are eliminated by detecting coding idioms discussed in Section 2.

The relevant information for FCA – elements, properties and their mapping – is generated as follows:

- The set of *elements* is built from the classes that are analyzed.
- The set of *properties* is built from the methods that are actually *understood* (see definition below) by all the classes contained in the set of elements. As a further optimization we do not include methods that are inherited from above the root of the hierarchy, *i.e.*, Object, because we normally do not want to include the class Object for refactorization.

  In addition to cancelled behavior we detect *duplication* of methods by comparing the decompiled method bytecode which lets us identify duplicated methods with the same behavior but possibly with different variable names or comments. In FCA each group of duplicated methods is treated as one single element in the properties set.
- The mapping between elements (classes) and properties (methods) is defined by a binary relation: a class C and a method m fulfill the relation R iff C implements or inherits the method m and any instance of class C *understands* m.

We define that a class *understands* a method iff the method can be executed without throwing an error indicating the method to be cancelled or abstract[4]. Although this property depends on the runtime behavior of the program, we are able to detect cancelled methods by a static analysis of the source code if we presume that (i) shouldNotImplement errors only occur within a sequence of self sends, *i.e.*, shouldNotImplement is never called when sending a message to another object than self and (ii) that a shouldNotImplement message is sent whenever the cancelled method is executed, *i.e.*, the shouldNotImplement statement is not within a conditional statement.

According to these assumptions we can test whether a method m is *understood* by an instance a of a class C by testing if (i) m includes a message send with the selector shouldNotImplement and (ii) all methods which can be invoked by messages sent from m to self or super, are *understood* by a. The second condition (ii) needs to perform a method lookup for each message which is sent to self or super starting at the class C or its superclass respectively. This strategy allows us to detect if a method is really understood by a class, *i.e.*, it is not (indirectly) cancelled.

4.2 Generating an Optimal Factoring

The main output produced by FCA in the previous step is the concept lattice. Each concept has a set of methods as properties, and a set of classes as elements. All the methods in a concept are (i) implemented by each of the classes (or by one of its superclasses) and (ii) *understood* by the instances of those classes.

Figure 4 shows a concept lattice produced by FCA for an input of the three concrete classes WS, RWS, and RS and a few selected methods. These classes (with abbreviated names) are the classes, WriteStream, ReadWriteStream and ReadStream which we discuss in Section 2 and which are illustrated by Figure 1. We add four methods to the methods already present in Figure 1 for additional illustration: three methods with the name contents which are implemented differently by each of the three classes (and thus named contents[1], contents[2] and contents[3]), and the method atEnd used by all three classes by inheriting from the common superclass.

The produced lattice shown in Figure 4 has three concepts with only one class and two concepts with two classes; the top concept has the full set of classes and the bottom concept has an empty set of classes. The actually understood methods of those sets of classes are listed in the lower part of the concept (the FCA properties of this concept). The top concept of the lattice has exactly one method, atEnd, which is shared by all classes. The bottom concept has an empty set of classes, because it shows which classes use all methods.

[4]In Smalltalk method abstractness or method cancellation are expressed as run-time errors raised by the execution of the methods doesNotUnderstand: and shouldNotImplement.

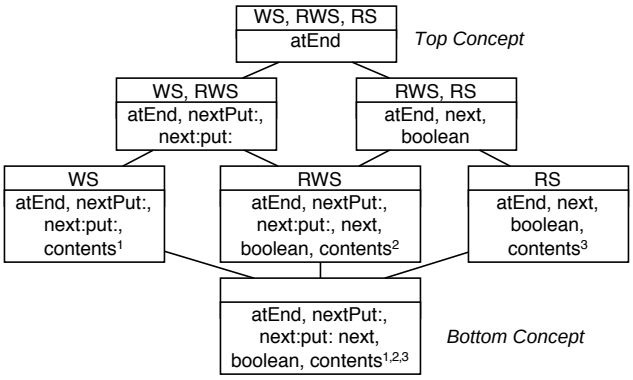

Figure 4: A lattice generated by FCA. Each concept, depicted as a rectangle, consists of a set of classes (upper part) and a set of methods (lower part).

In this second step of our approach we obtained:

- The exact set of methods that a class needs to implement, inherit or obtain from a trait.
- All the maximal subsets of these sets of methods shared between different classes.

4.3 Processing the Output of FCA

In addition to the related sets of classes and methods the lattice generated by the FCA engine provides the order of concepts. This means that each concept has one or more super- and subconcepts, except for the top and bottom concept. We use this information to reduce the sets of methods of a concept by the methods found in all its superconcepts. Hence, the concept only includes the so-called *gamma*, the methods that are additional in comparison to those of the superconcepts.

Figure 5 illustrates the applied reduction on the lattice of Figure 4.

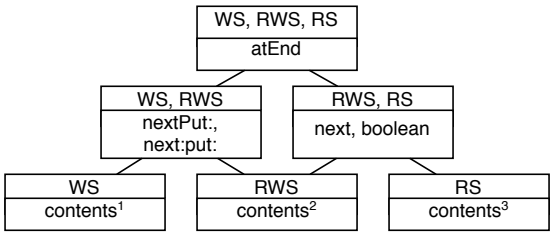

Figure 5: A gamma lattice generated by FCA.

In addition to reducing the methods, we remove the bottom concept because it does not have an interesting meaning for us. Now, most if not all of the concepts with only one class are located at the bottom level of the lattice. There is no concept at this level which does not have exactly one class because this would mean that there are methods which do not belong to a class or there are two classes that define exactly the same methods.

Naively, this lattice can be understood as a multiple inheritance class hierarchy with inheritance from top to bottom, each concept representing a class. If we analyze a particular class, its concept and all its superconcepts present groups of methods that, summed up, provide the full behavior of the class.

After the processing of the lattice as described above we obtain the following information:

- Each concept presents a maximal group of methods and a set of classes they belong to.
- Each bottom concept represents a class. Its set of (indirect) superconcepts in the lattice indicates which groups of methods are needed, either by inheritance or obtained from a trait, to get the full behavior.

This information is used in the next step to derive classes, traits and the new hierarchy.

4.4 Reconstructing the Hierarchy with Traits

In this step of the analysis the concepts are investigated to identify potential traits and to infer the new class hierarchy. The manual task of deciding whether a concept is a trait or a class is supported by our tool by providing relevant properties of the concepts such as *required* and *provided* methods or the number of classes. For example, we analyze the method body of the methods associated with a concept to identify required methods. The set of required methods of a concept are all messages that are sent to self but are not in the set of provided methods.

The following list provides some basic heuristics we developed while applying our tool in the case studies discussed in the subsequent Section 5.

- Concepts with only one class (in our example the three bottom concepts in Figure 5) define the concrete classes in the refactored system.
- The top concept, if its set of methods is not empty, is implemented as the root class of the hierarchy. In our previous example this would be a class that implements atEnd.
- Concepts providing many methods which originally belonged to the same abstract class, can again serve as abstract classes if possible. They may be further split into traits in the second stage of the process.
- Concepts with no or with just very few provided methods for which it is not worth creating a trait are not implemented as a class or trait. The methods of those concepts have to be pushed up in the hierarchy using the idiom of cancelling methods to make them available to the appropriate classes. This decreases the factoring of the hierarchy and again introduces rather undesired coding idioms. Nonetheless this rather pragmatic decision is, in our opinion, better than having to implement too many traits each defining just one or two methods. Also, this situation sometimes points out weaknesses in the original hierarchy. For example, methods that are not implemented in a class because they would have had to be duplicated. In these cases, a refactoring can solve this problem.
- The remaining concepts are good candidates for traits.

For a trait, its concept's elements *i.e.*, the set of classes, indicate which classes directly (or indirectly) need to apply the trait. The required methods show which methods need to be provided to be able to apply the trait. A small number of required methods indicates that the trait is more universally applicable in different contexts.

Applying the above principles in the example of Figure 5 leads to the following class hierarchy and traits. The bottom concepts become the concrete classes WS, RWS, and RS. The top concept can as well be implemented as a class because all three concrete classes use it. The remaining two concepts (the middle layer) are good traits. The new hierarchy would be implemented with the concrete three classes, inheriting from the root class and applying one (or in the case of RWS both) of the traits.

4.5 Input for FCA Invocation Analysis

In this step of the process we apply FCA again. This time we use a different setup and do not apply it to the whole hierarchy but to each individual class and trait we have identified in the previous step. The goal is to further decompose them into traits or subtraits where appropriate.

The set of elements, properties and the mapping for the FCA input is generated as follows. The set of elements are all methods defined locally by the class or trait we analyze. The properties are all possible method invocations of this class or trait. In other words, these are methods that can be invoked by sending messages to self within the class or trait.

For the relation between the elements (methods) and the properties (invocations), the *self-sends* of each method of the class are analyzed. With this information we identify for each method all the (transitively) invoked methods that are defined or inherited by the class. By transitively invoked methods we mean for example, if a method m1 defines exactly one *self-send* of m2 and m2 in turn one of m3, then the set of invocations of m1 is { m2, m3 }.

4.6 Decomposing Classes and Traits

The output produced by FCA in the previous step is a concept lattice which, without further processing, can be interpreted as follows. Each concept has as elements a maximal group of methods which all have the same set of invocations. These groups of methods are good candidates for traits because they are likely to be related as they invoke the same methods. The concept lattice provides the reengineer with several interesting alternatives for each potential trait: moving up the concept lattice reduces the set of common invocations (properties) and hence provides additional methods (elements) for bigger traits. In a similar way, if we walk down the lattice, the concepts have more invocations which further restrict the group of methods.

5. CASE STUDIES

To validate our approach we applied it to two significant inheritance hierarchies, namely the Smalltalk-80 Stream and Collection hierarchies [16], which are often considered to be paradigmatic examples of object-oriented design. These hierarchies have been the subject of case studies for the identification of interfaces [8, 2]. Godin *et al.* [15] also applied FCA to them. Moreover we chose this case study because the authors of traits manually refactored them to evaluate the relevance of traits [5]. Therefore by using the same case studies we strengthen the evaluation of our approach since we compare how our semi-automatic approach performs as compared to a *purely manual* approach.

In Squeak [17], the abstract class Collection has 98 subclasses, and the abstract class Stream has 39 subclasses, but many of these (like Bitmap or CompiledMethod) are special purpose classes and hence not categorized as "Collections" by the system organization. For the purposes of this study, we use the term Collection Hierarchy to mean Collection and its 37 subclasses that are also in the system category Collections. We use the term Stream Hierarchy to mean Stream and its 10 subclasses that are also in this category.

First we discuss the refactoring of the Stream Hierarchy, the results we obtained and then compare with the manual refactoring approach. Then we present an important part of the results we obtained by refactoring the Collection Hierarchy.

5.1 The new Stream Hierarchy

For this case study we chose to analyze the three most important concrete classes of the Stream Hierarchy, ReadStream, WriteStream and ReadWriteStream. Figure 1 shows the original inheritance relationship of these classes: ReadWriteStream is a subclass of WriteStream, WriteStream and ReadStream inherit from PositionableStream which finally inherits from Stream, the root of the hierarchy (simplified in Figure 1).

First stage: Hierarchy Analysis. The run of our tool to identify traits and restructure the hierarchy produced a total of nine concepts. After processing the output of FCA as described in Section 4.3 we now discuss the analysis of the obtained concepts to build the new hierarchy and traits. The rationale of this manual analysis are discussed in Section 4.4.

After filtering the concepts to reject the ones with less than two classes and less than two provided methods, we obtain three candidates for traits:

1. A concept with 42 methods shared between ReadStream and ReadWriteStream
2. A concept with 54 provided methods shared between WriteStream and ReadWriteStream
3. A concept with 36 methods and applying to all three classes we are analyzing

The first concept of the above list is a promising candidate for a trait. All methods implement behavior for *reading* from a stream. Separated into three groups these are: (i) next and similar methods like nextWord, nextInt32, reading the next object, word, or 32-bit integer from the stream, (ii) skipTo:, skipSeparators: etc. to move the access position and (iii) do:, iterating over the objects in the stream.

Most of these methods (37) are originally implemented in PositionableStream, three methods are implemented by Stream and two are originally defined in both ReadStream and ReadWriteStream, which means, that they have been duplicated.

The second concept is interesting as a trait as well. Having the users WriteStream and ReadWriteStream, its 54 methods define the behavior to *write* on a stream. There are three main groups of behavior: (i) nextPut:, nextPutAll:, nextWordPut: etc. to write on objects, collections, words on the stream, (ii) cr, tab, space and friends for conveniently writing carriage returns, spaces etc. and (iii) growTo:, pastEndPut:, position:, reset, setToEnd to grow the collection and setting the write position and write limit.

From those methods WriteStream originally implemented 32 methods, PositionableStream 19 methods and Stream 3 methods.

The third and last concept of the above list has all three classes as elements and thus we would rather implement it as a common superclass of the three classes than as a trait. An additional argument against using a trait is that most of the methods originate either in PositionableStream or Stream; both are common superclasses of the three concrete classes we analyze.

Figure 6 illustrates the refactored Stream Hierarchy. The two traits we identified in this first stage of the refactoring are named TReadableStream and TWriteableStream. As discussed in Section 4 we create the three concrete classes from the methods found in the corresponding concepts. In our solution, we retain the two

abstract classes PositionableStream and Stream; the methods they implement are the ones from the remaining third concept we identified above. ReadWriteStream which was formerly a subclass of WriteStream (see Figure 1) is now a direct subclass of PositionableStream, and is using both traits TReadableStream and TWritableStream. By using traits to share reading and writing behavior, PositionableStream nolonger defines behavior which is partly cancelled in subclasses. Thus, its number of methods decreased from 84 to 22.

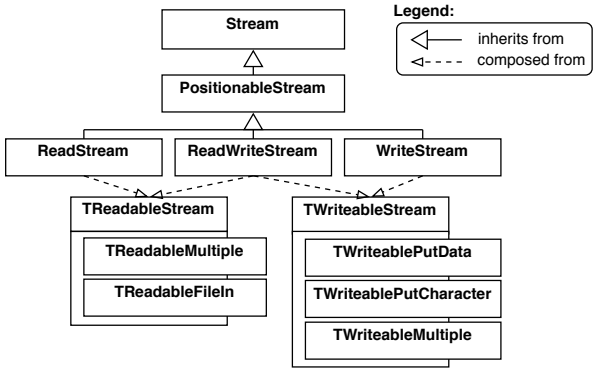

Figure 6: The refactored Stream Hierarchy with seven identified traits.

Second stage: Traits Decomposition. On each of the traits and classes we identified above we apply the analysis to detect fine-grained traits as described in step 5 and 6 of our approach (see Sections 4.5 and 4.6).

Applied to TReadableStream, FCA produced a total of 27 concepts. We chose two concepts to be implemented as subtraits:

- TReadableMultiple which provides five methods that implement retrieving multiple elements at once from a stream, (*e.g.*, next:into: which reads a number of elements specified by the first argument and stores them in the collection provided as second argument).
- TReadableFileIn which provides eight methods (such as fileIn, nextChunk or skipStyleChunk) concerned with reading source code from a stream which was previously "filed out".

For the trait TWriteableStream we retrieved the following subtraits:

- TWriteablePutData which implements writing data on the stream. The trait provides six methods such as uint16:, uint24:, uint32:, string: etc.
- TWriteablePutCharacter which implements convenience methods to write characters on the stream. Methods, from a total of seven, are cr, tab, crtab, space.
- TWriteableMultiple with nine methods. It provides the means to write multiple objects on the stream at once: next:putAll:, string:, verbatime: and timestamp and methods that write source code on the stream: nextChunkPut: or nextPutKeyword:withArg:.

Thus, with the last stage of our refactoring we were able to decompose the two traits which we obtained from the first stage into more fine-grained traits. The new traits are applied as subtraits to the original traits as Figure 6 shows.

Comparison with the manual refactoring. Our semi-automatic refactoring of the Stream Hierarchy succeeded in producing a very similar result compared to that of the manual refactoring approach [5]. The main restructuring of making ReadWriteStream a direct subclass of PositionableStream and sharing reading- and writing-behavior between the three concrete classes with traits, is identical with their solution. Our two main traits, TReadableStream and TWritableStream, match with only very few exceptions the provided methods of the corresponding traits in their solution.

The manual refactoring differs in respect to how the two main traits are further split up into subtraits. Their solution has exactly one subtrait for each of the two traits to group behavior that is not related to the positionability of a stream. For example, the behavior to read from a stream is defined in the traits TReadablePositionable and its subtrait TReadable.

Our approach is also capable of identifying some subtraits but it does not distinguish methods by the property of positionability. Also, our decomposition into subtraits seems to be only partial successful: for example no trait in TWriteableStream was detected for filing-out code, nor is there a trait for reading single characters.

5.2 The new Collection Hierarchy

We applied our approach on the most important concrete classes of the Collection Hierarchy: Array, OrderedCollection, SortedCollection, Set. Figure 7 shows a part of the original inheritance hierarchy.

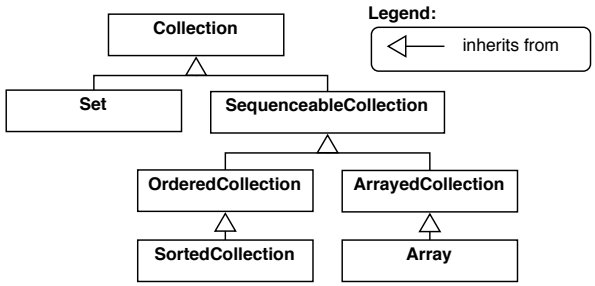

Figure 7: Part of the original Collection Hierarchy

Identified traits and new inheritance hierarchy. We now describe part of the derived new Collection Hierarchy which is illustrated by Figure 8. The refinement with subtraits is not discussed here because of space reasons. The main changes are that composition of behavior through trait use replaces subclassing and cancellation as in case of SortedCollection which was previously a subclass of OrderedCollection and now is of SequenceableCollection but shares the trait TOrderedSortedCommon. This change to the hierarchy and the new trait TOrderedSortedCommon improves the factoring because no unwanted behavior, *e.g.*, adding elements at a specific position in the collection, is inherited and thus no methods have to be cancelled.

Another case where the inheritance relationship changes are the classes OrderedCollection and Array which share a trait named TSequencedElementAccess. This trait is not used by SortedCollection due to the fact that a sorted collection cannot mutate an object at a specific position. In the original implementation this behavior was prevented by cancelling methods like at:put: or addFirst: in SortedCollection. Hence, SortedCollection cannot inherit from OrderedCollection anymore - we make it a direct subclass of SequenceableCollection as Figure 8 illustrates. Another trait we identified and named TOrderedSortedCommon, implements common behavior of

OrderedCollection and SortedCollection. There are still methods implemented locally in OrderedCollection. This is the behavior that is neither shared with SortedCollection nor with Array: these are methods that implement adding elements at a specific position because Array cannot add elements and SortedCollection cannot add elements at a specific place because it is sorted explicitly. Our approach detects those methods in the concept which has as element only the class OrderedCollection. The methods are: add:after:, add:before:, addFirst: etc. and some methods that are specific to the implementation of OrderedCollection such as at:put: and collect:.

Another trait we identified is used by all classes except Array and Dictionary. Array cannot remove objects and Dictionary uses a different interface (removeKey:). Originally the methods for removing were pushed up to Collection. Our tool proposes to factor them out into a trait and use them where appropriate. The trait has the requirement remove:ifAbsent:, originally declared in Collection as a abstract hook method. This method is defined by most classes in a different way according to their kind of collection. As Figure 8 shows, we named the trait TRemovingElements and apply it to all classes except Array and Dictionary (latter is not shown in Figure 8). OrderedCollection and SortedCollection indirectly use the trait via the subtrait of TOrderedSortedCommon.

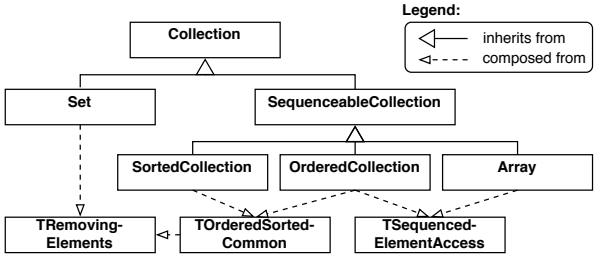

Figure 8: Part of the refactored Collection Hierarchy

Comparison with the manual refactoring. Compared to the manual refactoring approach the most obvious difference is the number of identified traits (although, we restricted our analysis to only the most important classes): our refactoring produced a couple of traits whereas their solution includes 40 traits. We now compare the three traits which were discussed above with the traits in the manual refactoring.

The trait TSequencedElementAccess has similar methods as the trait TElementAccessSM in the manual refactoring. But our solution found two additional methods for this trait, swap:with: and integerAt:put:. On the other hand our solution does not implement methods for sorting the collection. The reason for this is that originally those methods were not understood by OrderedCollection but only by Array and hence our algorithm does not detect them although they would be applicable for an ordered collection. This problem of missing implementation was identified in Section 2 as a symptom of missing traits use.

The trait TOrderedSortedCommon of our solution (see Figure 8) is named identically as in the manual refactoring and many of the 36 methods in our trait can be found in their trait.

The trait TRemovingElements corresponds to the trait TExtensibleU in the manual refactoring. The difference is that our trait does not implement methods for adding elements because this behavior is implemented differently by the users of TRemovingElements. The reason why this behavior is nevertheless implemented in the manual refactoring is because their traits are organized in different layers where traits override inappropriate behavior from subtraits. In contrast, our approach does not have any distinction between implementation- and interface-traits as they have.

The difference between our solution produced by FCA and the manual refactoring is big in respect to the number of traits. The reason why the manual refactoring proposes a lot more traits is because it modularizes the primitive properties more finely than would have been necessary to avoid code duplication. The intent of the manual refactoring was to create a hierarchy in which new reuse possibilities would be possible.

Our approach is *curative* in the sense that it identifies traits in existing code to support better reuse *i.e.*, to get rid of code duplication or the need to implement methods too high in the hierarchy. The manual refactoring is *speculative* as especially stressed by Black *et al.*, to apply fine-grained traits for better understandability and reusability by users outside of the hierarchy.

The experience proved us that our approach produced a valuable, although rather minimal solution. It showed to be able to identify the different aspects, *e.g.*, implicit/explicit ordering or extensibility of Array, OrderedCollection and SortedCollection and to propose a reasonable solution including a restructuring of the inheritance hierarchy.

6. DISCUSSION

The first stage of our approach identifies maximal groups of common methods for a set of classes and hence it only detects traits based on implementation sharing situations between classes. This produces a maximally factored inheritance hierarchy with traits for sharing behavior that would not be possible with single inheritance. From the point of view of reuse it does not make sense to further decompose those traits. But there are reasons for grouping behavior into more fine-grained traits, *e.g.*, for better understandability of code, or for potential reuse at a later point. Thus, our approach has a second stage to decompose those traits by analyzing method invocations.

Our case studies showed that we recognized traits that were identified on the same code by manual analysis. One important property of our approach is that it not only infers traits from an existing sharing situation, *e.g.*, inheritance, but also succeeds in identifying traits even when the actual code is not shared.

To identify the actual methods a class should implement, inherit or obtain from a trait, our algorithm detects cancelled and duplicated methods. Whereas cancellation was widely used in the hierarchies we analyzed, duplication was only detected in few cases. The reason is that duplication is widely accepted as bad practice and the hierarchies we studied are stable and well-designed hierarchies. Moreover our algorithm only detects methods that have the same decompiled bytecode which means that their code only differs in respect to variable names or comments. An interesting enhancement of our approach would be to factor our common expressions of methods into new methods to increase reuse possibilities.

Limitations. The analysis of our case studies shows also the limitations of our approach. Our tool detects less traits compared to the manual refactoring which aimed at achieving a maximal decomposition [5]. The general problem is that our approach obviously does not take the conceptual meaning of methods into account because it is based on the analysis of structural relationship between methods and classes. For example, we do not necessarily distinguish between different meanings of methods such as collect: and select: versus add: and remove: unless they can be detected by analyzing the invocation relationships – but this is not always the case. Another example of such a conceptual distinction in the manual refactoring is the separation of traits into interface- and implementation layers which were not detected by our approach.

7. RELATED WORK

In the context of inheritance hierarchy analysis and restructuring, FCA was used in different ways.

Godin *et al.* [15] developed incremental FCA algorithms to infer implementation and interface hierarchies that are expected to have maximal factorization, *i.e.*, they are guaranteed to have no redundancy. To analyze their solutions from a point of view of complexity and maintainability they propose a set of structural metrics. As experiment they also analyze the Smalltalk Collection Hierarchy. Their mapping for FCA is very similar as our setup for the first stage of the analysis. One important limitation is that they consider each method declaration as a different method and thus cannot identify code duplication as in our case. Moreover their approach serves rather as a help for program understanding than reengineering since the resulting hierarchies cannot be implemented in Smalltalk because of single inheritance.

Interfaces and specifications of the Smalltalk Hierarchy were also analyzed by Cook [8]. He also takes method cancellation into account to detect protocols. By manual analysis and development of specifications of the Smalltalk Collection Hierarchy he proposes a better protocol hierarchy. Protocol hierarchies explicitly represent similarities between classes based on their provided methods. Thus, compared to our approach, protocol hierarchies present a *client* view of the library rather than one of the *implementor*. Former does not reveal the implementation of a hierarchy and thus does not serve for our purpose of detecting traits.

Moore [18] proposes automatic refactorization of inheritance hierarchies in Self [25], although not using FCA. Like with our approach, the structure of the original hierarchy is irrelevant in his tool and refactoring can discover new relationships between classes that did not exist before. In contrast to our approach, Moore focuses on factoring out common expressions in methods. In the resulting hierarchies none of the methods and none of the expressions that can be factored out are duplicated. Moore's factoring creates methods with meaningless names which can be a problem if the code should be read.

In Casais' approach [7] he uses an automatic structuring algorithm to reorganize class hierarchies in Eiffel using decomposition and factorization. In his approach, he increases the number of classes in the new refactored class hierarchy. In our case, we keep the same number of classes or even remove abstract classes but add traits in the new refactored class hierarchy.

The approaches by Godin, Moore and Casais are similar to our approach in that they analyze inheritance hierarchies to propose better factorization. One important difference in comparison to our approach is that we do not only take method declarations into account but also invocation relationships. Hence, our approach is additionally capable of detecting even fine-grained groups of methods that can be factored out for better reusability and program understandability.

In C, Snelting and Tip analyzed a class hierarchy making the relationship between class members and variables explicit [22]. By analyzing the *usage* of the hierarchy by a set of client programs they were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class.

Also from the perspective of a set of client programs which use the hierarchy, Streckenbach *et al.* infer improved hierarchies in Java with FCA [23]. Similar to our tool, their proposed refactoring can then be used for further manual refactoring. Their editor allows the reengineer to change the proposed result as long as it stays consistent and it is able to generate code. If not, the tool proposes the reengineer to move up methods in the hierarchy to work around multiple inheritance generated by the generated lattice.

The work of Streckenbach is based on the analysis of the usage of the hierarchy by client programs. The resulting refactoring is behavior preserving (only) with respect to the analyzed client programs. Our approach differs in that we do not analyze the hierarchy from a client view, producing optimal factoring in respect to the *purpose* of the hierarchy in specific scenarios, but rather from the *implementation* point of view which features a globally behavior preserving refactoring: each concrete class exhibits the identical behavior as the original one.

8. CONCLUSIONS AND FUTURE WORK

We presented a semi-automatic approach for the identification of traits in an existing class hierarchy. Our tool proposes a refactoring of the class hierarchy with traits which preserves the original behavior of each of the classes. The approach is based on FCA and is two-staged: it is based (i) on a static analysis of source code, detecting idioms in Smalltalk which are commonly used to work around restrictions of single inheritance and (ii) on an analysis of the invocation relationships between methods to detect fine-grained traits. The case study shows that the results we obtained were similar — as far as the intention of the method grouping was the same – to the ones obtained manually.

An interesting enhancement to improve our approach will be to do method refactoring as mentioned in Section 6 and proposed by Moore [18]. Another area we like to investigate into is to detect traits based on the usage of the classes of a library by analyzing applications that use it.

Acknowledgments. 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), and Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

9. REFERENCES

[1] G. Arévalo. Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis. In *Proceedings of LMO '03*, pages 47–59. Hermes, Paris, Jan. 2003.

[2] A. Arfi, R. Godin, H. Mili, G. W. Mineau, and R. Missaoui. Generating the interface hierarchy of a class library. In *Colloquium on Object Orientation in Databases and Software Engineering*, pages 42–57, 1994.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[4] A. P. Black and N. Schärli. Traits: Tools and methodology. In *Proceedings of ICSE 2004*, pages 676–686, May 2004.

[5] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of OOPSLA'03*, volume 38, pages 47–64, Oct. 2003.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[7] E. Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, Dec. 1994.

[8] W. R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92*, volume 27, pages 1 – 15. ACM Press, Oct. 1992.

[9] A. Dunsmore, M. Roper, and M. Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00*, pages 467–476. ACM Press, 2000.

[10] K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, Dec. 2003.

[11] The fortress language specification. research.sun.com/projects/plrg/fortress0618.pdf.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

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

[15] 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.

[16] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., 1983.

[17] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, 1997.

[18] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96*, pages 235–250. ACM Press, 1996.

[19] Scala home page. http://lamp.epfl.ch/scala/.

[20] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of ECOOP 2003*, pages 248–274. Springer Verlag, July 2003.

[21] C. Smith and S. Drossopoulou. Chai: Typed traits in java. In *Proceedings ECOOP 2005*, 2005.

[22] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[23] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *Proceedings of OOPSLA '04*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[24] D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings of ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

[25] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.

[26] N. Wilde and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

Formal Concept Analysis in a Nutshell

Formal concept analysis (FCA) [14] is a branch of lattice theory that allows us to identify meaningful groupings of "elements" that have common "properties" (referred to, respectively, as *objects* and *attributes* in the standard FCA literature[5]).

To illustrate FCA, let us consider a toy example about musical preferences. The *elements* are a group of people *Frank*, *Anne*, *Arthur*, *John*, *Thomas*, and *Michael*; and the *properties* are *Rock*, *Pop*, *Jazz*, *Folk*, and *Tango*[6]. The following table shows which people prefer which kind of music, and is called the *incidence table*

prefers	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: Incidence Table of the Music Example

A *context* is a triple $C = (E, P, R)$, where E and P are finite sets of elements and properties, respectively, and R is a binary relation between E and P represented in the incidence table. In the musical preferences example, the elements are the people, the properties are the different kinds of music they prefer, and the binary relation *prefers* is defined by Table 1. For example, the tuple *(Frank, Folk)* is in R, but *(Anne, Jazz)* is not.

Let $X \subseteq E$, $Y \subseteq P$, and $\sigma(X) = \{p \in P | \forall e \in X : (e, p) \in R\}$ and $\tau(Y) = \{e \in E | \forall p \in Y : (e, p) \in R\}$, then $\sigma(X)$ gives us all the *common properties* of the elements contained in *X*, and $\tau(Y)$ gives us the *common elements* of the properties contained in Y, *e.g.*, $\sigma(\{$*Arthur, Catherine*$\}) = \{$*Jazz*$\}$.

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 $X = \tau(Y)$. In other words, a concept is a maximal collection of elements sharing common properties. In Table 1, a concept is a maximal rectangle we can obtain with relations between people and musical preferences. For example, *(*$\{$*Frank, Anne*$\}$, $\{$*Rock, Pop*$\}$*)* is a concept, whereas *(*$\{$*Catherine*$\}$, $\{$*Jazz*$\}$*)* is not, since $\sigma(\{$*Catherine*$\})$ = $\{$*Jazz*$\}$, but $\tau(\{$*Jazz*$\})$ = $\{$*Arthur, Catherine, Thomas, Michael*$\}$. The extent and intent of each concept is shown in the following table.

top	({ all elements },∅)
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	(∅,{ all properties })

Table 2: The set of concepts of the Music example

The set of all the concepts of a given context forms a *complete partial order*. Thus we define that a concept (X_0, Y_0) is a **subconcept** of concept (X_1, Y_1), denoted by $(X_0, Y_0) \leq (X_1, Y_1)$, if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$). Inversely we define that the concept (X_1, Y_1) is a **superconcept** of concept (X_0, Y_0). For example, the concept $(\{Anne\},\ \{Rock, Pop, Tango\})$ is a subconcept of the concept $(\{Frank, Anne\},\ \{Rock, Pop\})$. Thus the set of concepts constitutes a *concept lattice* $\mathcal{L}(\mathcal{T})$ and there are several algorithms for computing the concepts and the concept lattice for a given context. For more details, the interested reader should consult Ganter and Wille [14].

[5]We use the terms *element* and *property* instead to avoid the unfortunate clash with object-oriented terminology.

[6]The full property names are *prefers Pop* or *prefers Folk*. We abbreviate these names for the sake of conciseness.