Discovering Unanticipated Dependency Schemas in Class Hierarchies *

Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz
Software Composition Group, University of Bern
www.iam.unibe.ch/∼scg

Abstract

Object-oriented applications are difficult to extend and maintain, due to the presence of implicit dependencies in the inheritance hierarchy. Although these dependencies often correspond to well-known schemas, such as hook and template methods, new unanticipated *dependency schemas occur in practice, and can consequently be hard to recognize and detect. To tackle this problem, we have applied* Concept Analysis *to automatically detect recurring dependency schemas in class hierarchies used in object-oriented applications. In this paper we describe our mapping of OO dependencies to the formal framework of Concept Analysis, we apply our approach to a non-trivial case study, and we report on the kinds of dependencies that are uncovered with this technique. As a result, we show how the discovered dependency schemas correspond not only to good design practices, but also to "bad smells" in design.*

Keywords: Concept Analysis, Class Hierarchies, Schemas.

1. Introduction

Inheritance is the cornerstone of object-oriented development, enabling conceptual modeling, subtype polymorphism and software reuse. But inheritance can be used in subtle ways that make complex systems hard to understand and extend [16]. In particular, a developer making changes or extensions to an object-oriented system must understand the implicit contracts and dependencies between a class and its subclasses, or risk that seemingly innocuous changes break these contracts [13][12].

Many dependencies in well-designed object-oriented systems are not *ad hoc*, but rather correspond to well-known idioms, coding conventions and design patterns. Other dependencies may be signs of weak programming practices. In either case, implicit, undocumented dependencies lead to fragile class hierarchies that are difficult to extend or modify correctly.

Although an experienced software developer may know what to look for, it can be very tedious and time-consuming to search for undocumented dependencies in a large and complex software base. Automated tools can help to search for certain expected types of dependencies, but unfortunately the set of possibilities is open-ended.

Formal Concept Analysis (FCA) is a technique to identify recurring *concepts* amongst a set of *elements* with certain *properties*. A "concept" is a set of properties exhibited by a given number of elements. These concepts are then related to one another in the form of a lattice. We apply FCA to the problem of uncovering implicit dependencies in class hierarchies used in object-oriented software systems. To apply this technique, we map method *invocations* and attribute (*i.e.*, instance variable) *accesses* to FCA elements, and consider different predicates over invocations and access as FCA properties. The concepts that are identified correspond to a set of invocations and accesses whose properties characterize what we call *dependency schemas* — recurring sets of dependencies over methods and attributes in a class hierarchy.

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

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

Uncovered identified dependency schemas may correspond either to well-known best practice in object-oriented design, or they may be signs of degenerated design. Once dependency schemas are classified, they can be a good basis for identifying which parts of a system are in need of repair. The approach thus provides us not only with a *global* view

* In Proceedings of CSMR 2005 (9^{th} European Conference on Software Maintenance and Reengineering), IEEE Computer Society, pp. 62–71, 2005

of the system and which kinds of dependencies and practices occur, but it also provides *detailed* information about how specific classes are related to others in their hierarchy, and how that hierarchy can be modified and extended.

The paper is structured as follows: Section 2 presents an overview of the architecture of our approach. In Section 3 we provide a summary about the main definitions of FCA. In Section 4 we present our mapping of object-oriented dependencies to the framework of FCA. In Section 5 we provide an overview of the results obtained by applying our approach to the *Smalltalk Collection* hierarchy and by showing how SortedCollection fits into this hierarchy. Section 6 provides a brief discussion of various technical issues. Section 7 presents some related work. Finally, in Section 8 we conclude with some remarks on future work.

2. Overview of the Approach

Our approach is based on a *pipeline* architecture in which the analysis is carried out by a sequence of processing steps (Figure 1). The output of each step provides the input to the next stage. We have implemented the approach as an extension of MOOSE reengineering environment [6]. In this section we give a brief overview of each step.

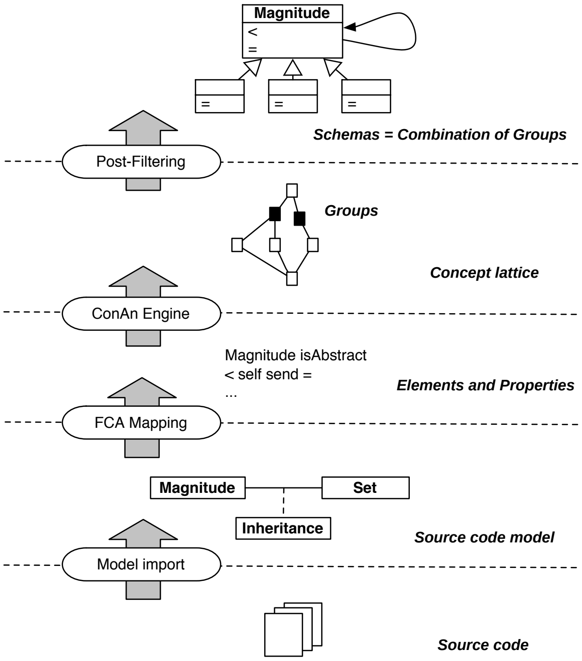

Figure 1. The overall approach

Model Import. The first step is to build a *model* of the application from the source code. For this purpose we use the MOOSE reengineering platform [6], a research vehicle for reverse and reengineering object-oriented software. Software models in MOOSE conform to the FAMIX metamodel [14], a language-independent metamodel for reengineering.

FCA Mapping. In the second step, we map the model entities to *elements* and *properties*, and build a *incidence table*, which shows which elements fullfil each property, so that we can apply FCA. This mapping is described in detail in Section 4.

ConAn Engine. ConAn is our implementation of an FCA engine. Once the elements and properties are defined, we run ConAn to build the concepts and the lattice. ConAn applies the Ganter algorithm [7] to build the concepts and our own algorithm to build the lattice [2].

Post-Filtering. After the concepts and the lattice are built, each concept consists of a group of invocations and accesses with commonalities representing a *candidate* for a *dependency schema*. But not all the concepts are relevant. Thus we have a *post-filtering* process, which is the last step. Thus, we filter out meaningless or uninteresting concepts.

Analysis We analyze *candidate* concepts representing the detected *dependency schemas* –resulting from the previous steps – and we use them to explore how the hierarchy has been built based on these schemas.

This approach is not specific to this case study. It was applied to case studies similar to detect *unanticipated* dependencies between different software entities in an application [4][3].

3. Formal Concept Analysis in a Nutshell

Formal concept analysis (FCA) [7] 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[1]).

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*[2]. Table 1 shows which people prefer which kind of music, and is called the *incidence table*

A *context* is a triple $C = (E, P, R)$, where E and P are finite sets of elements and properties, respectively, and R is

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

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

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 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 Table 2.

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 [7].

4. FCA Mapping: Setup of Formal Context

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

4.1. Elements and Properties of Classes

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

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

- *i is an invoked method in C* [3]
- *i is an accessed attribute in C* [4]
- *C invokes m via self*
- *C invokes m via super*
- *C defines m*
- *C defines a*
- *C_1 is ancestor of C*
- *C_1 is descendant of C*
- *m is abstract* in C
- *m is concrete* in C
- *m is cancelled* in C [5]

[3] *i is an invoked method in C* means that there is a method in C that calls method i

[4] *i is an accessed attribute in C* means that there is a method in C that accesses attribute i

[5] A method m in class C is *cancelled* in Smalltalk when it calls the selector `shouldNotImplement` to indicate that the class C wants to undefine the method m defined in a superclass.

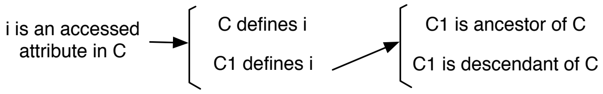

Figure 2. Possible combination of properties applied to accessed attributes.

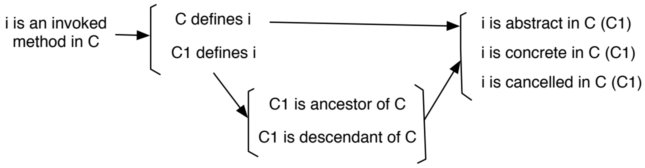

Figure 3. Possible combination of properties applied to invoked methods.

Firstly, we directly adopt the predicates *C invokes* m *via self* and *C invokes* m *via super* as properties, then we combine the rest of predicates to obtain the final list of properties. Not all the combinations are valid. Figures 2 and 3 show all the valid combinations to get the properties ('→' and '[ ]' represent the logical symbols *and* and *or* respectively). The obtained properties are[6]:

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

For example, if we take the first three properties we build them in the following way:

- i *accesses local state* = i *is an accessed attribute in C* and *C defines* i
- i *accesses state in Ancestor* C_1 *of C* = i *is an accessed attribute in C* and C_1 *is ancestor of C* and C_1 *defines* i
- i *accesses state in Descendant* C_1 *of C* = i *is an accessed attribute in C* and C_1 *is descendant of C* and C_1 *defines* i

The rest of the properties are built in a similar way.

4.2. Interpretation of the Properties in Concepts

By applying FCA to this context, we obtain a set of concepts and certain of them correspond to interesting *dependency schemas*. In our analysis, *the conjunction of properties of the intents of the concepts* determines if we have found a potential *meaningful dependency schema*. Let us briefly consider two examples reported in Section 5: *Reuse of Superclass Behavior* and *Broken* super *send Chain* . Just to clarify how the concepts are interpreted we list in both cases the set of elements and properties contained in each concept. But to avoid figures with confusing information, they illustrate some elements (as an example of all elements contained in the concepts) participating in the schema.

The example of schema *Reuse of Superclass Behavior* (shown in Figure 5) is interpreted with the following elements and properties:

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

Another possible example is the schema *Broken* super *send Chain* (shown in the Figure 7) and it is composed of the following elements and properties:

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

5. Case Study Results

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

[6] The notation used is the same as cartesian product, meaning that {a,b}x{c,d} = {(ac),(ad),(bc),(bd)}

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

Table 3. Commonly Identified Schemas.

We first show our own categorization of detected schemas, then provide a *global* overview of the schemas discovered, and then we focus on the role of the class SortedCollection within the collection.

5.1. Detected Dependency Schemas

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

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

Table 3 provides an overview of the three groups, together with the total number of detected instances of each schema. Due to space limitations, we do not describe each schema in detail. We only discuss a few examples of each category. In each case, we show which are the set of properties that allows us to infer the schema.

5.2. Global View on Collection Hierarchy

As we said in the Section 1, the *dependency schemas* provides us with a basis to understand a class hierarchy identifying where the recurring kinds of dependencies relating classes. In this section, we introduce some of the schemas used in a global analysis of a class hierarchy.

Classical*: Local Direct State Access.*

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

FCA Properties: $\{i$ *accesses local state* $\}$.

"Bad Smell"*: Ancestor Direct State Access.*

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

FCA properties: $\{i$ *accesses state in Ancestor* C_1 *of* C $\}$

"Bad Smell"*: Cancelled Local or Inherited Behavior.*

Description: This schema identifies concrete or local inherited methods that are invoked via a *self* send in a class or its superclasses but are then cancelled in subclasses. Method cancellation is a sign that inheritance is being applied purely for purposes of code reuse, without regard for subtyping. Since methods of the superclass calling the cancelled methods can still be called on the cancelling class, this may lead to runtime errors. In the Collection hierarchy it occurs in the subhierarchies of SequenceableCollection and OrderedCollection.

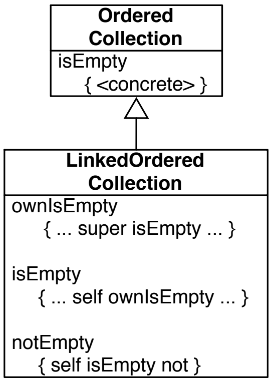

Figure 4. Inherited and Local Invocations

FCA properties: $\{C$ *invokes* i *via self*, i *is concrete locally*, i *is cancelled in descendant* C_1 *of* C $\}$ or $\{C$ *invokes* i *via self*, i *is concrete in ancestor* C_1 *of* C , i *is cancelled in descendant* C_2 *of* C $\}$

Irregularities*: Inherited and Local Invocations.*

Description: This schema shows methods that are invoked by both *self* and *super* sends within the same class. Initially this schema is a *good practice* coding, but a problem occurs when the *super* sends are invoked from a method with a different name. This special case of the schema occurs in the classes LinkedOrderedCollection, LinkedWeakAssociationDictionary and XMainChangeSet. All these classes have a special form: the class overrides a method m and m invokes a method named own-m via *self* send, and this last method calls m via a *super* send implemented in the superclass. Figure 4 illustrates this schema. This is an *irregular* case of the schema *Redefined Concrete Behavior* because the class is overriding the superclass behavior but is indirectly using the superclass behavior.

FCA properties: $\{C$ *invokes* i *via self*, C *invokes* i *via super*, i *is concrete locally*, i *is concrete in ancestor* C_1 *of* C $\}$

5.3. “Class-Based” View on SortedCollection

So far, we have introduced some schemas that help us to analyze a class hierarchy from a *global view*, but our approach helps also us to analyze how a class is built in the context of its superclasses and subclasses, meaning from *local view* regarding a class.

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

In this class we identify 12 different schemas that involves this class. Again for space reason, we present 4 of them belonging to different categories:

Classical*: Reuse of Superclass Behavior.*

Description: This schema shows us that the class SortedCollection calls via *self* the methods copyEmpty, insert:before:, reverseDo:, asArray, isEmpty, notFoundError and they are not defined in the class but different superclasses define their behavior. Specifically, we see that the methods copyEmpty, insert:before: and isEmpty are defined in the class OrderedCollection, reverseDo: and asArray are defined in the class SequenceableCollection; and notFoundError is captured in the class Collection. Thus, we see which are the superclasses that determine the behavior of the class. Figure 5 illustrates this schema.

“Bad Smell”*: Broken* super *send Chain.*

Description: This schema identifies methods that are extended (*i.e.*, performing a *super* send) in a class but redefined in their subclasses without calling the overridden behavior, hence giving the impression to break the original extension logic. In SortedCollection the methods = and representBinaryOn: are invoking superclass hidden methods. But the definition of these methods in the subclass SortedCollectionWithPolicy does not invoke the method defined in SortedCollection. Such a behavior can lead to unexpected results when the classes are extended without a deep knowledge of them. Figure 7 illustrates this schema.

FCA Properties: $\{C$ *invokes* i *via super*, i *is concrete locally*, i *is concrete in descendant* C_1 *of* C , i *is concrete in ancestor* C_1 *of* C $\}$

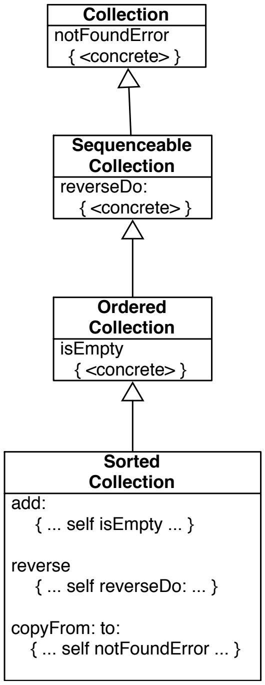

Figure 5. Reuse of Superclass Behavior.

“Bad Smell”*: Cancelled Local Behavior but Superclass Reuse.*

Description: This schema shows that the method addLast: is called *via a super send* and this method is defined in the immediate superclass OrderedCollection, meaning that the class is reusing the behavior of the superclass. But this method is also implemented in the class SortedCollection but the behavior is *cancelled.* Although it is not a good practice, it seems a normal situation because the elements in a *sorted collection* cannot be added in the end of the collection, but in a predefined position defined by the sorting function of the class. As we said previously, this is a case where the inheritance is used as code reuse without regarding *subtyping*. Specifically, this means that SortedCollection is a kind of OrderedCollection but not all the inherited methods can be applied. Figure 6 illustrates this schema.

FCA properties: $\{C$ *invokes* i *via super*, i *is concrete in ancestor* C_1 *of* C , i *is cancelled locally* $\}$

Irregularities*: Inherited and Local Invocations.*

Description: This schema shows that the method copyEmpty is used with *self* sends and *super* sends in the class *SortedCollection.* It is implemented in the class itself, has an implementation in the superclass Collection and an implementation in the subclass SortedCol-

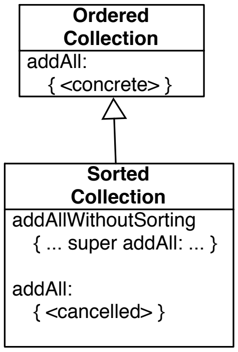

Figure 6. Cancelled Local Behavior and Behavior Reuse of Superclasses.

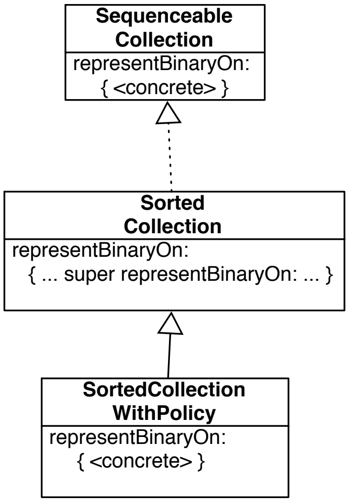

Figure 7. Broken *super* send Chain.

lectionWithPolicy. When checking the code, we see that the most of the calls are *self* sends and in the method called copyEmpty, we have a *super* send to a method with the same name. This means that, in spite of a local implementation of copyEmpty, finally the behavior of this method is determined by the superclasses, showing a heavy reuse of the superclass code. Figure 8 illustrates this schema.

FCA properties: $\{$*C invokes i via self*, *C invokes i via super*, *i is concrete locally*, *i is concrete in ancestor* C_1 *of C* $\}$

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

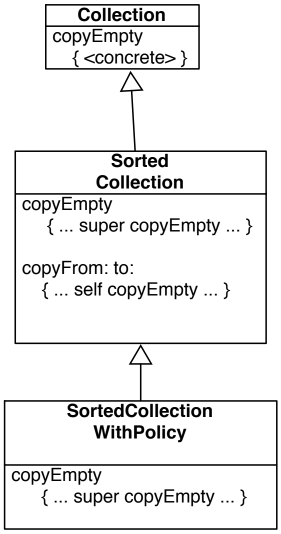
Collection
copyEmpty
{ <concrete> }
Sorted Collection
copyEmpty
{ ... super copyEmpty ... }
copyFrom: to:
{ ... self copyEmpty ... }
SortedCollection WithPolicy
copyEmpty
{ ... super copyEmpty ... }

Figure 8. Inherited and Local Invocations.

6. Discussion

Use of FCA: FCA offers the possibility of specifying simple properties between the elements we need to analyze. Based on these simple properties, as we have seen in the Section 2, the FCA algorithm (to build the concepts) implemented in ConAn provides us with all the possible combinations of elements with a common set of properties. In this way we “mine” the *concepts* we use to identify possible dependency schemas in the class hierarchies. This approach allows us to detect not only well-known, but also *unanticipated* relationships between the different elements. Thus, we are able to identify all possible combinations of properties that could characterize a schema actually used in a class hierarchy.

Performance of the Algorithm: In Section 4 we saw that we need to map the model entities (in our specific case, the invocations and accesses) to *CA elements* and build different properties based on them. Due to a limitation imposed by FCA algorithm (to build the concepts) in performance measurements [10], we reduce the amount of CA elements to compute the concepts and the lattice without losing information about the class hierarchies. Thus, if we have several invocations of the same method or several accesses to the same attribute in the same class, we keep only one invocation or one access per class as a representative, and we reduce dramatically the number of CA elements used by

ConAn, and improved the computing time from around 1 hour to 10 min compared to the approach presented in an earlier paper [1].

Analysis of State and Behavior: There are two main differences with regard to our previous work [1]. First, in the current paper we take dependencies to state into account, whereas our earlier work considered only behaviour. This yields more concepts, and hence more schemas of interest than when only behaviour is considered. Secondly, we are able to categorize schemas into those that represent *good, irregular* and *bad* design decisions in the class hierarchies.

Partial Usage of Lattice: In Section 4, we pointed out that once the concepts and the corresponding lattice are built, each concept represents a group of invocations and accesses that relate a group of classes. But not all the concepts are relevant, and we keep only the meaningful concepts for our analysis. There are main two points worth mentioning. Firstly, we only use 64 of 174 concepts in total, meaning that the 1/3 of the lattice represents meaningful information in our approach from our analysis. Finally, we must note that, in this particular application of FCA, we do not use the *partial order* of the concepts in the lattice. This means that we do not exploit the possible relationships between the schemas (mapped from the concepts).

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

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

FCA vs. Logic Engine: One of the main results of this approach is a *catalog* of schemas to characterize a class hierarchy. As we see in Section 4.2, each schema is the interpretation of a *conjunction* of properties in the concepts. Then, each schema can be expressed as a logic predicate (mapped from the properties) and a logic engine can be run in a class hierarchy to identify the occurrences of the different schemas. Thus, we must remark that the main difference between the use of FCA and a logic engine is that in the first case, we do not know in advance which are the possible schemas occurring in the class hierarchies, and consequently we do not know the combination of properties that characterize them. FCA helps us mainly to discover *unpredictable* schemas introduced in a class hierarchy. In the case of the use of a logic approach, we must know which are the different properties that characterize a schema. We consider that both approaches are complementary ones, because the catalog of schemas can be complete after the analysis of several class hierarchies, and in that moment, the use of logic engine is most suitable than FCA.

7. Related Work

Various researchers have explored techniques to support the understanding and evolution of class hierarchies using FCA. We briefly summarize their works.

Dekel uses CA to visualize the structure of the class in Java and to select an effective order for reading the methods and reveal the state usage [5]. Godin and Mili [8] uses concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. In C++, Snelting and Tip [11] 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. As a result, they propose a new class hierarchy that is behaviorally equivalent to the original one. Similarly, Huchard [9] applied concept analysis to improve the generalization/specialization of classes in a hierarchy. Tonella and Antoniol [15] use CA to detect the *structure* of Gamma-style design patterns using relationships between classes, such as *inheritance* and *composition*.

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

8. Conclusion and Future Work

In this paper, we show how the automatic generation of schemas using FCA helps us to discover different implicit and undocumented dependencies in class hierarchies in terms of the behavior and state usage. The categorization of these schemas into *good*, *irregular* and *bad smell* design decisions allows us to localize where different irregularities or problems occur in the implementation of a class hierarchy. With the *dependency schemas* we are able to analyze a class hierarchy from a global viewpoint considering all the classes, and from a local viewpoint considering considering one class and the relationships with its subclasses and superclasses.

We have previously explored two other applications of FCA. With *X-Ray Views* [4], we analyze a class in isolation (without the context of superclasses and subclasses). The study is based on the use of behavior and the state, and the collaborations of different defined methods inside the classes. This approach complements the information of the schemas Local Behavior and *local state* introduced in this paper.

We have also previously used FCA to explore behavioral dependencies [1] in class hierarchies. In that approach, we analyze only the relationships between classes based on the behavioral dependencies (*self* and *super* sends and where the methods are captured). We introduced a possible list of schemas that show the different design decisions in a class hierarchy. In this paper, we complement the information with state accesses and also analyse the cancellation mechanism of methods. We also provide a categorization of the schemas to be able to concentrate on the irregularities and problems in the class hierarchies.

We plan to apply the approach to other class hierarchies to check if the catalog of schemas identified thus far covers all the interesting possible cases or if we discover new cases of implicit contracts. Another interesting analysis is the combination of the methods that are invoked and those that are not invoked but are declared in the classes. This kind of approach can measure how much information defined in the class hierarchy is used or not. Another feature to improve is the refinement of properties to be able to get a mapping 1 to 1 from concept to schema, and thus, reduce the complexity of the lattice in terms of number of concepts. Last but not less important is the analysis of relationships given by the *partial order* between the different schemas -mapped from the concepts in the lattice.

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, Oct. 2002 - Sept. 2004).

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] G. Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Jan. 2005.

[3] G. Arévalo, F. Buchli, and O. Nierstrasz. Detecting Implicit Collaboration Patterns. In *Proceedings of WCRE '04*, pages 122–131. IEEE Computer Society Press, Nov. 2004.

[4] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray Views: Understanding the Internals of Classes. In *Proceedings of ASE '03*, pages 267–270. IEEE Computer Society Press, Oct. 2003.

[5] U. Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, Feb. 2003.

[6] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

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

[8] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

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

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

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

[12] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA '95*, pages 200–214. ACM Press, 1995.

[13] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96*, pages 268–285. ACM Press, 1996.

[14] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings of ISPSE '00*, pages 157–167. IEEE Computer Society Press, 2000.

[15] P. Tonella and G. Antoniol. Object Oriented Design Pattern Inference. In *Proceedings of ICSM '99*, pages 230–238. IEEE Computer Society Press, Oct. 1999.

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