Understanding Classes using XRay Views *†

Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz
Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern
3012 - Bern, Switzerland
{arevalo, ducasse, oscar}@iam.unibe.ch

Abstract

Understanding the internal workings of classes is a key prerequisite to maintaining an object-oriented software system. Unfortunately, classical editing and browsing tools offer mainly linear and textual views of classes and their implementation. These views fail to expose the semantic relationships between the internal parts of a class. We propose XRay *views —a technique based on Concept Analysis— which reveal the internal relationships between groups of methods and attributes of a class. XRay views are composed out of elementary collaborations between attributes and methods and help the engineer to build a mental model of how a class works internally. In this paper we present XRay views, and illustrate the approach by applying it to three Smalltalk classes: OrderedCollection, Scanner, and UIBuilder.*

Keywords: Class Understanding, Concept Analysis, Logical Views

1 Introduction

It is well-established that 50% to 75% of the overall cost of a software system is devoted to maintenance [20]. During maintenance, it is estimated that software professionals spend at least half their time reading and analysing software in order to understand it [7]. Code reading is not only a means to understand software, but is also a viable strategy for checking and assuring software quality [3]. These facts show that understanding source code is a key activity in the maintenance of software systems.

Although object-oriented technology was expected to reduce the cost of developing and maintaining large software systems, paradoxically the contrary is the case. On the one hand, object-oriented techniques help engineers master complexity, so object-oriented systems tend to last long, and thus enter their "maintenance" phase much earlier. On the other hand, object-oriented systems can be harder to understand than procedural systems [9, 18], so the cost of maintenance can actually be higher. This is due to several reasons [26, 5, 10]:

1. Contrary to procedural languages, the method definition order in a file is not important [9]. There is no simple and apparent top-down call decomposition, even if some languages propose the visibility notion (private, protected, and public). Furthermore, the run-time architecture is not apparent from the source code, which only exposes the class hierarchy [10].

2. The presence of late-binding leads to "yoyo effects" when walking through a hierarchy and trying to follow the call-flow [26].

We propose a technique to support software engineers in the task of understanding a complex object-oriented system. Instead of requiring the engineer to read code line-by-line, we provide three logically connected "XRay views" of classes that give the engineer an impression of the relationships between methods, attributes, and the invocation and access patterns. In this way we support *opportunistic* understanding [22] in which the engineer understands a class iteratively by exploring patterns and reading code. To be precise in the rest of the paper, we use the term *collaboration* to express a relationship between a set of methods and a set of attributes.

Let us take a simple example. Imagine we want to understand how the (Smalltalk) class OrderedCollection works. In VisualWorks 7.0 its source code contains the following comment:

*This paper has been accepted as a short paper in the ASE 2003 Main Conference

†In Proceedings of 2nd International MASPEGHI 2003 Workshop colocated in ASE 2003 (Montreal, Canada), pp. 9-18, CRIM - University of Montreal, 2003

"OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. Elements are accessible by external keys that are indices. OrderedCollections can act as stacks or queues."

OrderedCollection has two named attributes, firstIndex and lastIndex and an indexed attribute (*i.e.*, an array-like attribute). OrderedCollection inherits 377 methods and implements 56 methods of which 25 redefine inherited methods. Simply reading the source code of these methods would clearly be a slow process that would not necessarily reveal the collaborations between the three attributes and the methods of this class.

Instead, an XRay view will tell us that the methods notEmpty:, size, remove:ifAbsent:, includes:, find:, at:, and at:put: *systematically* access the attributes firstIndex *and* lastIndex. Moreover the class does not define any setter or getter methods for these attributes and we can consequently infer that the methods mentioned previously are responsible for maintaining the class invariant. Armed with this information, the engineer can now focus attention on these methods and gain further insight into the workings of this class.

Generalizing from this example, here is a (non-exhaustive) list of the kind of information that an engineer would typically like to know about a class:

- which methods access any attribute, directly or indirectly
- which groups of methods access directly or indirectly all the attributes or some subset of the attributes,
- which methods are only called internally,
- which methods/attributes are heavily used and accessed,
- how the methods and attributes collaborate.

Each of these aspects is important for understanding the inner workings of a class, but unfortunately they are implicit in the source code, and therefore cannot easily be teased out by a straightforward reading of the source. For this reason we generate a graph representation of the source code and run our tool, ConAn, which applies *Concept Analysis* to detect different collaborations to compose them in the XRay views. In this paper we limit our approach to understanding a single class, without taking into account relationships to subclasses, superclasses, or peer classes.

Structure of the Paper. In Section 2 we provide a brief introduction to Concept Analysis. In Section 3 we show how to interpret the collaboration of methods and attributes within classes using Concept Analysis. In Section 4 we show how the concepts identified help us to interpret different XRay views of classes. In Section 5 we present an overview of related work, and in Section 6 we summarize the results so far and outline our future work.

2 Concept Analysis in a Nutshell

Concept analysis (CA) [12] (also known as Concept Galois lattices [27]) is a branch of lattice theory that allows us to identify meaningful groupings of "objects" that have common "attributes". (NB: To avoid confusion with object-oriented terminology, we refer in this paper instead to *elements* having common *properties*.)

To illustrate CA, 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*[1]. Table 1 shows which people prefer which kind of music.

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. Elements and their satisfied properties in 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. 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

$$\begin{aligned} \sigma(X) &= \{p \in P | \forall e \in X : (e,p) \in R\} \\ \tau(Y) &= \{e \in E | \forall p \in Y : (e,p) \in R\} \end{aligned}$$

$\sigma(X)$ gives us all the *common attributes* of the elements contained in *X*, and $\tau(Y)$ gives us the *common objects* of the properties contained in Y. For example, $\sigma(\{$*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

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

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 $\sigma(\{\textit{Catherine}\}) = \{\textit{Jazz}\}$, but $\tau(\{\textit{Jazz}\}) = \{\textit{Arthur, Catherine, Thomas, Michael}\}$. The extent and intent of each concept is shown in Table 2.

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

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

The set of concepts forms a partial order known as a *concept lattice*. There are several algorithms for computing the concepts and the concept lattice for a given context [16, 21]. For more details, the interested reader should consult Ganter and Wille [12].

3 Applying Concept Analysis to Class Understanding

Complex software systems are composed of a large number of different kinds of entities (classes, methods, modules, subsystems) and a lot of different kinds of relationships that hold between them. CA can help us to detect patterns in these relationships, but first we must encode the software information at hand in terms of elements and properties. Depending on exactly what kinds of patterns we are interested in, we may apply CA in radically different ways.

In this paper we apply CA to identify concepts that correspond to the collaborations within a single class. We therefore choose as elements the *methods* and *attributes* of a class, and as properties the *access* and *invocation* relationships between them.

3.1 Elements and Properties of Classes

Suppose a class has a set of methods $\mathcal{M}$ and a set of attributes $\mathcal{A}$. The basic properties we use are extracted from the source code as follows:

- m *reads* x means that the method $m \in \mathcal{M}$ either directly reads the value of attributes $x \in \mathcal{A}$ or uses a "getter" method to access the value of x. We call m a *reader* method of x.
- m *writes* x means that the method $m \in \mathcal{M}$ either directly updates the value of attribute $x \in \mathcal{A}$ or uses a "setter" method to modify the value of x. We call m a *writer* method of x.
- m *calls* n means that the method m calls the method n explicitly via a *self-call*.

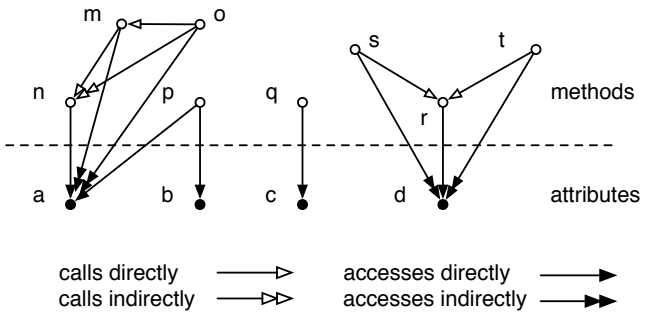

Figure 1. Attribute accesses and method invocations and the groups they form.

We also define a number of derived properties, for example:

- m *accesses* x if either m *reads* x or m *writes* x (*i.e.*, *accesses* $=$ *reads* $\cup$ *writes*)

In Figure 1 we see a graphical representation of a class with methods $\mathcal{M} = \{m, n, o, p, q, r, s, t\}$ and attributes $\mathcal{A} = \{a, b, c, d\}$. Here we have o *calls* m, m *calls* n, n *accesses* a, and so on.

These properties express direct relationships between entities. We are also interested in *indirect* relationships, for example, m *accesses* a *indirectly* (which we write "m *accesses** a"). Indirect relationships are important in revealing collaborations between methods and attributes, and are helpful in assessing the impact of changes. We therefore define as well the following derived properties:

- m *calls** n if m *calls* m' and either m' *calls* n or m' *calls** n (*i.e.*, *calls** $= \cup_{i \geq 2}$ *calls* ^{i})
- m *reads** x if m *calls* m' or m *calls** m', and m' *reads* x (*i.e.*, *reads** $= \cup_{i \geq 1}$ *calls* $^i \cdot$ *reads*)
- m *writes** x if m *calls* m' or m *calls** m', and m' *writes* x (*i.e.*, *writes** $= \cup_{i \geq 1}$ *calls* $^i \cdot$ *writes*)
- m *accesses** x if m *reads** x or m *writes** x (*i.e.*, *accesses** $=$ *reads** $\cup$ *writes**)

In the example, we see that o *calls** n and n *reads* a, and consequently o *reads** a.

We apply CA to our example class to reveal the following concepts: $(\{m, o\}, \{\textit{accesses}^* \ a\})$, $(\{n\}, \{\textit{accesses} \ a\})$, $(\{p\}, \{\textit{accesses} \ a, b\})$, $(\{q\}, \{\textit{accesses} \ c\})$, $(\{s, t\}, \{\textit{accesses}^* \ d\})$ and $(\{r\}, \{\textit{accesses} \ d\})$.

Finally, we are sometimes interested to know when elements do *not* exhibit a certain property, so we introduce the following notation to express the negation of a relation:

- $e \neg R \ p$ if it is not true that $e \ R \ p$.

For example, $o \neg$ *reads* a.

3.2 Collaborations

Since we are interested in collaborations occurring between *sets* of methods and attributes, we extend our properties to sets in the obvious way. Suppose that F and G are arbitrary subsets of the set of elements E. We define:

- $F \ R \ G$ means that each entity in F is related with each one in G, *i.e.*, $\forall e \in F, e' \in G, e \ R \ e'$.
- $F \ \overline{R} \ G$ means that the entities in F are *related exclusively* with those in G, *i.e.*, $\forall e \in E, e' \in G, e \ R \ e', \Rightarrow e \in F$ and conversely, $\forall e \in E, e' \in F, e' \ R \ e \Rightarrow e \in G$.

3.3 Interpretation

We introduce now the collaborations based on which XRay views are built. Note that in each case we are interested in *all* of the participants of a given collaboration. For example, below we define Collaborating Attributes, but we are interested not only in the attributes themselves, but also in the set of methods that access them. This holds for each example collaboration listed below.

Direct Accessors: Direct accessors, readers or writers $M \subseteq \mathcal{M}$ of an attribute a are defined by non-exclusive relationships:

- M *accesses* $\{a\}$
- M *reads* $\{a\}$
- M *writes* $\{a\}$

This collaboration provides us with a simple classification of the methods according to which attributes they use. In our example, $\{n, p\}$ *accesses* $\{a\}$.

Exclusive Direct Accessors: A method m is an *exclusive direct accessor* of a when m is the *only* method to access a directly. We are interested in the sets of exclusive direct accessors of an attribute:

- $M \ \overline{\textit{reads}} \ \{a\}$
- $M \ \overline{\textit{writes}} \ \{a\}$

In our example, we see that $\{r\} \ \overline{\textit{accesses}} \ \{d\}$.

Exclusive Indirect Accessors: We consider a method to be an *exclusive indirect accessor* when it calls a *direct accessor* method of a specific attribute. It is represented as an exclusive relationship:

- $M \ \overline{\textit{accesses}^*} \ \{a\}$
- $M \ \overline{\textit{reads}^*} \ \{a\}$
- $M \ \overline{\textit{writes}^*} \ \{a\}$

This collaboration helps us to distinguish those methods that define the behaviour of a class without using at all the state from those that use the state of the class. In our example, we have $\{s, t\} \ \overline{\textit{accesses}^*} \ \{d\}$.

Collaborating Attributes: This collaboration expresses which attributes are used exclusively by a set of methods:

- $M \ \overline{\textit{reads}} \ A$
- $M \ \overline{\textit{writes}} \ A$

In the example, we have the sets of attributes accessed exclusively by sets of methods are all of size 1: $\{q\} \ \overline{\textit{reads}} \ \{c\}$ and $\{r\} \ \overline{\textit{accesses}} \ \{d\}$.

Stateful Core Methods: This collaboration is a special case of *collaborating attributes* and expresses which methods access *all* the state of a class:

- $M \ \overline{\textit{reads}} \ \mathcal{A}$ and
- $M \ \overline{\textit{writes}} \ \mathcal{A}$

This collaboration is interesting because it provides a guideline if all the attributes are collaborating in the core of the class, and providing a functionality to the class through a set of methods. In the example, there are no methods accessing the entire state of the class.

Collaborating Methods: This collaboration expresses which methods uses the behaviour defined in the class. It is represented by an *exclusive dependency*:

- $M \ \overline{calls^*} \ M'$
- $M \ \overline{calls} \ M'$

This collaboration helps us to identify the direct and indirect collaborations between groups of methods inside the class. In the example, $\{o\} \ \overline{calls} \ \{m\}$, $\{m\} \ \overline{calls} \ \{n\}$, $\{o\} \ \overline{calls^*} \ \{n\}$ and $\{s,t\} \ \overline{calls} \ \{r\}$.

Interface Methods: This collaboration expresses which methods are not used at all inside the class. It is represented with an *exclusive dependency* as:

- $\mathcal{M} \ \overline{\neg \, calls} \ M$

M is the complete set of interface methods since there is no method in $\mathcal{M}$ that calls them, and there exist no other such methods.

$\mathcal{M} \ \overline{\neg \, calls} \ \{o,p,q,s,t\}$ identifies the interface methods of Figure 1.

Externally Used State: This dependency expresses which interface methods are *direct accessors*:

- $\mathcal{M} \ \overline{\neg \, calls} \ M$ and M *accesses* $\{a\}$

This collaboration helps us to determine which methods are used as interface to the class and access directly the state of the class. In the example, p and q provide externally used state, since $\mathcal{M} \ \overline{\neg \, calls} \ \{p\}$ *accesses* $\{b\}$ and $\mathcal{M} \ \overline{\neg \, calls} \ \{q\}$ *accesses* $\{c\}$.

Stateless Methods: This collaboration expresses which methods complement the *collaborating* ones, *i.e.*, which methods provides a service without calling any other methods or accessing the state of the class:

- $M = M_1 \cap M_2$, where $M_1 \ \overline{\neg \, calls} \ \mathcal{M}$ and $M_2 \ \overline{\neg \, accesses} \ \mathcal{A}$

There are no stateless methods in the example, since every method either calls another method or accesses some state.

4 XRay Views

An XRay view is a *group* of collaborations that exposes specific aspects of a class. Based on the collaborations specified above, we now define three XRay views: STATE USAGE, EXTERNAL/INTERNAL CALLS, and BEHAVIOURAL SKELETON. These three views address different, but logically related aspects of the behaviour of a class. STATE USAGE focuses on the way in which the state of a class is accessed by the methods, and exposes, for example, how cohesive the class is. EXTERNAL/INTERNAL CALLS categorizes methods according to whether they are internally or externally used, while BEHAVIOURAL SKELETON focuses on the way methods invoke each other internally.

In order to illustrate our approach, we present three Smalltalk classes —OrderedCollection, UIBuilder, and Scanner— from the VisualWorks Smalltalk distribution [25]. We chose these particular three classes because they are different enough in terms of size and functionality, they address a well-known domain that the reader is certainly familiar with, and they show characteristic results of XRay view application. Here follows a brief description of these classes:

OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. The elements are accessible by external keys that serve as indices. This class has attributes firstIndex and lastIndex that index the first and last elements in the collection. Moreover OrderedCollection has an anonymous array-like attribute. Its behaviour is defined by 56 methods from which 24 redefine methods inherited from the superclass.

UIBuilder implements the Builder design pattern [1]. It is a complex class that is used to build user interfaces (windows and their subcomponents) according to declarative specifications provided by its clients. A UIBuilder is created and used at interface opening time by the client's interface opening method. UIBuilders use a special library of user interface components tailored for automatic user interface generation such as radio buttons, action buttons, and check boxes. UIBuilders can build and install composites of these components to any desired level of nesting. This class has 18 attributes and its behaviour is defined in 122 methods.

Scanner represents a traditional language scanner for the Smalltalk language. It scans a stream of Smalltalk tokens with a single look ahead. This class has ten attributes which refer to the source, to the current character, current token, current token type, a type table, and comments. Its behaviour is defined with just 24 methods which are procedurally-coded.

We describe the three XRay views according to a common pattern: first we provide a *description*, then the *collaborations* used to build the view, and finally a *rationale* indicating the key aspects that the view can reveal. Note that the results provided by different views will often overlap, as the views provide different perspectives of the same class using the same elements.

Class	HNL	Attributes	Methods
OrderedCollection	3	3	56
UIBuilder	1	18	122
Scanner	1	10	24

Table 3. Data about the classes (HNL indicates the level of inheritance

Due to space limitations, we describe just the first view, STATE USAGE, in detail, and only summarize the remaining views. For each view, we ran our analysis tool, ConAn, on the three classes, we examined the resulting views by looking at and combining the groups presented in the "Used and Shown Collaborations" section of the view definition, and we validated our findings by reading the source code opportunistically.

4.1 XRay View: STATE USAGE

Description: Clusters attributes and methods according to the way methods access the attributes.

Used and Shown Collaborations: Exclusive Direct Accessors, Exclusive Indirect Accessors, Collaborating Attributes, and Stateful Core Methods.

Rationale: Objects bundle both public and private behaviour and state. In order to understand the design of a class, it is important to gain insight into how the behaviour accesses the state, and what dependencies exist between groups of methods and attributes.

Validation with OrderedCollection: As shown in Figure 2, STATE USAGE leads to the following groups.

- Exclusive Direct Accessors
  - {before, removeAtIndex:, add:beforeIndex:, first } $\overline{reads}$ {firstIndex} and {removeFirst, removeFirst:, addFirst } $\overline{writes}$ {firstIndex },
  - {after, last } $\overline{reads}$ {lastIndex } and {removeIndex:, addLastNoCheck:, removeLast, addLast:, removeLast: } $\overline{writes}$ {lastIndex }
- Exclusive Indirect Accessors
  - {addLast:, copyWithout:, select:, trim, add:, representBinaryOn:, add:before:, increaseCapacity, collect:, grow, after:, add:after:, addAllLast:, addAll:, addAllFirst:, removeLast: } $\overline{accesses}$ {firstIndex }
  - {copyWithout:, select:, add:beforeIndex:, add:, addFirst:, trim, add:, representBinaryOn:, removeAtIndex:, collect:, increaseCapacity, grow, removeFirst:, add:before:, before:, add:after:, addAllLast:, addAll:, addAllFirst:, removeLast: } $\overline{accesses}$ {lastIndex }
- Collaborating Attributes,
  - {makeRoomAtFirst, changeSizeTo:, removeAllSuchThat:, makeRoomAtLast, do:, notEmpty:, keysAndValuesDo:, detect:ifNone:, changeCapacityTo:, isEmpty, size, remove:ifAbsent:, includes:, reverseDo:, find:, setIndices, insert:before:, at:, at:put:, includes: } $\overline{accesses}$ {firstIndex, lastIndex }
- Stateful Core Methods = the same set as Collaborating Attributes

Before analysing the groups identified by this view, we posed the hypothesis that the two attributes maintain an invariant representing a memory zone in the third anonymous attribute.

First, we note that the attributes firstIndex and lastIndex have no getters or setters, so the state of the class is not exposed to clients.

Second, by browsing Exclusive Direct Accessors methods, we confirm the hypothesis that the method removeFirst accesses firstIndex and removeLast: accesses lastIndex.

The numbers of methods that exclusively access each attribute are very similar, however, we discover that firstIndex is mostly accessed by readers, whereas lastIndex, is mostly accessed by writers.

It is worth noting that Collaborating Attributes are accessed by the same methods that are identified as Stateful Core Methods. This situation is not common even for classes with a small number of attributes, and reveals a cohesive collaboration between the attributes.

We identified 20 over 56 methods in total that access systematically *all* the state of the class. By further inspection, we learned that most of the accessors are readers. There are only five methods, makeRoomAtFirst, makeRoomAtLast, setIndices, insert:before:, and setIndicesFrom:, that read and write the state at the same time. More than half of the methods (33 over 56) directly and indirectly access both attributes.

This confirms the hypothesis that the class maintains a strong correlation between the two attributes and the anonymous attribute of the class.

Validation with UIBuilder: The results are quite different compared to those obtained for OrderedCollection.

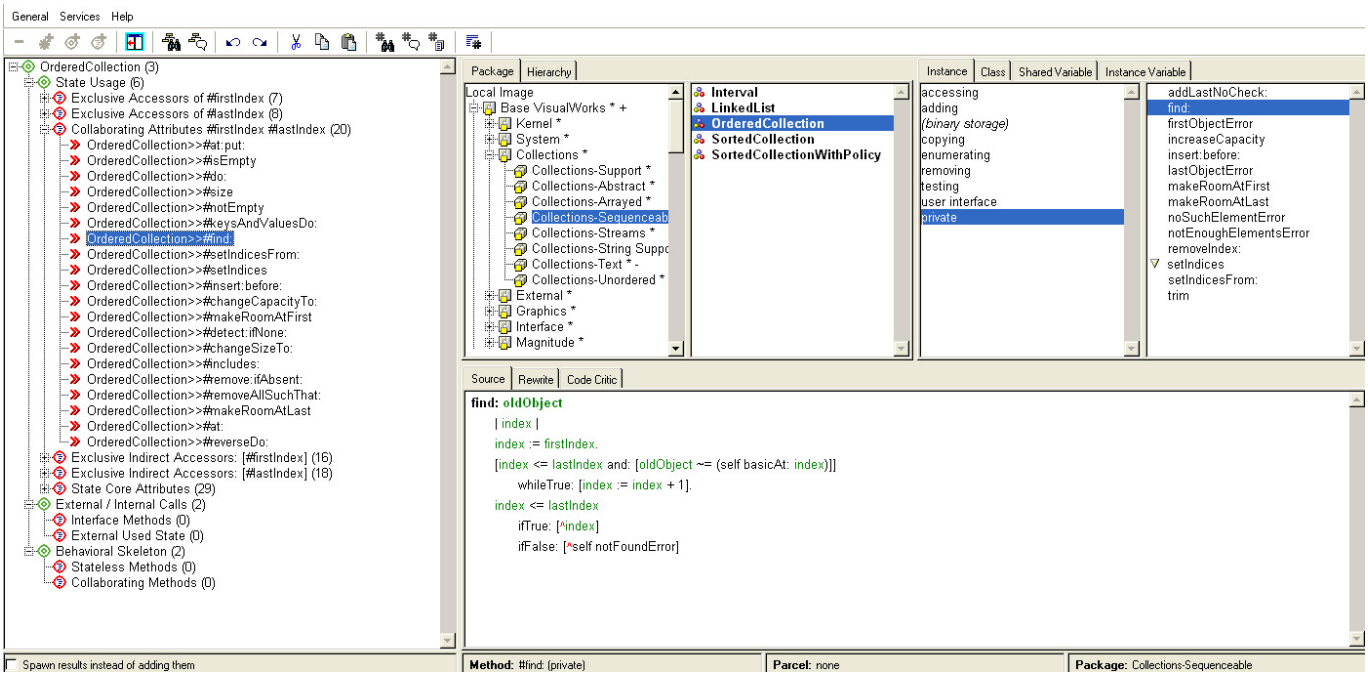

Figure 2. An XRay **view** STATE USAGE **applied to the class** OrderedCollection**. Here we browse the** Collaborating Attributes **collaborations and the method** find: **of this collaboration.**

First, we find getters and setters for each attribute. If we consider only the methods that access directly the attributes, we can classify the attributes into three groups:

- a) attributes that are accessed only through their getter and setter (policy, stack, cacheWhileEditing, and decorator);
- b) attributes that are accessed through their getter and setter, and an additional method (labels, values);
- c) attributes that are accessed by several methods. Note that the view EXTERNAL/INTERNAL CALLS helps us to refine our understanding of these differences.

We also learned that most accessors are readers, and there are only very few writers. Most of the writer methods are setters. This means that most of the attributes either are initialized when instances are created or are initialized and modified outside the class scope.

If we consider the collaborations among the attributes taking into account only the direct accessors, we find that there are very few groups of collaborating attributes: (wrapper, component), (bindings, window), (stack, composite), (policy, window), (source, bindings), (component, decorator, wrapper). The methods access groups of attributes only by reading them. 9 over 18 attributes are used with other ones. This means that there are 9 attributes that are used alone in different methods, so this fact reveals that the class is grouping several functionalities and could be split using the set of non-collaborating and collaborating attributes. This kind of hypothesis can be refined using the BEHAVIOURAL SKELETON view.

When we look at indirect accesses to attributes we obtain some new groups of collaborating attributes but these new groups only include two *new* attributes that were not identified by the direct access attribute groups. From this observation we can learn that there is a group of 11 core attributes that are used in the same group of methods.

In this specific case, we do not have any stateful core methods, which is not surprising as the class has a lot of attributes.

Validation with Scanner: The results for Scanner are completely different from those obtained for the other two classes. We find that we cannot partition the attributes into groups that are exclusively used by certain sets of methods. Instead, each method typically uses some subset of attributes that overlaps in arbitrary ways with those used by other methods. This means that every attribute offers some specific functionality that is complemented by the functionality offered by other attributes. None of the attributes have

setters and getters, *i.e.*, the state is internal and it is not exported outside the scope of the class.

4.2 XRay View: EXTERNAL/INTERNAL CALLS

Description: Clusters methods according to their participation in internal or external invocations.

Used and Shown Collaborations: Interface Methods and Externally Used State.

Rationale: This view reveals the overall shape of the class in terms of its internal reuse of functionality. This is especially important for understanding framework classes that subclasses will extend. Interface methods, for example, are often generic template methods, and internal methods are often hook methods that should be overridden or extended by subclasses.

Validation with OrderedCollection: OrderedCollection has 37 external methods. Of these, there are 22 methods that directly access attributes. Therefore the class OrderedCollection has a flat call-flow which means that there is little internal reuse of its own behaviour.

The groups also reveal that on the one hand we have methods such as add:, remove: that are part of the public class interface but are also used internally, and on the other hand we have pure, public methods such as changeSizeTo: and representBinaryOn:.

In VisualWorks, the class OrderedCollection has 6 subclasses. However, each of these subclasses only *adds* extra behaviour and does not change the internal behaviour of the class. This confirms our expectations, since the absence of internal reuse of methods in OrderedCollection is also a sign that there is little behaviour to be reused or extended by subclasses.

Validation with UIBuilder: 89 of 124 UIBuilder methods are not invoked by the class itself. Just 31 methods define the internal behaviour of the class. This fact fits well with the intent of the Builder design pattern and the fact that UIBuilder offers not only a lot of functionality to build complex user interface but also offers several ways to query its internal state via methods such as componentAt:, listAt:, and menuAt:.

Moreover we checked how the accessor methods identified by the STATE USAGE are classified as external and internal methods. policy and decorator are external, as they allow the client of the builder to specify the look and feel policy used for the window. Curiously, the attribute stack is simply not used, whereas cacheWhileEditing is purely internal, as its name suggests.

Note that this is a typical example how different views like STATE USAGE and EXTERNAL/INTERNAL CALLS complement each other in the process of understanding a class.

Validation with Scanner: In the case of Interface Methods, we have a group composed of two Smalltalk method categories[2] of Scanner: multi-character scans and public interface. Within the first group, we can find methods such as xDigit or xLetter that are used to form numbers or words, and within the second group, we have the methods defined as *public access*: {breakIntoTokens:, scanFieldNames:, scanPositionsFor:inString:, scanTokens: }. Thus we see that the view reveals a traditional narrow interface of a scanner, and it confirms the hypothesis that we present in the view STATE USAGE that all the methods and the attributes are collaborating inside the class.

4.3 XRay View: BEHAVIOURAL SKELETON

Description: Clusters methods according to whether or not they collaborate with other methods defined in the class or whether or not they access the state of the class.

Used and Shown Collaborations: Collaborating Methods and Stateless Methods.

Rationale: Ideally an object should be cohesive. In reality, this is not always the case. For example user interface classes usually act as a glue between the domain objects and the widgets. The way methods form clusters of collaborating methods indicates whether a class is cohesive or not [4].

Validation with OrderedCollection: We do not observe much collaboration among the methods. The collaborations between them are made in pairs, and there are not so many *calls** , *i.e.*, the method add:after: calls the method find:, and this last method does not call any other method in the class. Most of the methods confirm this fact, *i.e.*, that there are no groups of methods collaborating with other groups. We saw in the STATE USAGE view that most of the methods access the attributes of the class. Now we can confirm that the methods do not collaborate with each other, thus illustrating the flat method call-flow pattern for this class. For stateless methods, we identify two main groups: (1) those that access a global variable, and (2) those that invoke methods, such as error:, inherited from the superclass.

[2]Method categories in Smalltalk are groups of methods without language semantics that support code browsing.

Validation with UIBuilder: In the EXTERNAL/INTERNAL CALLS view we see that UIBuilder has 31 methods that constitute the internal behaviour. We see that the call-flow is a complicated structure, and the internal collaborations are more complex than in the other two classes.

Validation with Scanner: In this class, we identify a situation similar to that with OrderedCollection. The collaboration between the methods occurs in pairs, and there are no groups of methods collaborating with other groups. Since Scanner is a small class, it is not surprising that the internal collaborations are simple.

5 Related Work

We find different kinds of applications for understanding object-oriented systems at the class level both using CA and using other approaches.

Among those that use CA, we have two main axes: one for analysing individual classes and the other one for analysing class hierarchies.

Dekel uses CA to visualize the structure of the class in Java and to select an effective order for reading the methods [9]. He calculates all the accesses to fields that each method makes. In contrast to our approach, the concept lattice he calculates does not provide information about the interaction between the methods, nor does it reveal whether a method accesses a combination of fields directly, by accessing their values, or indirectly, by invoking methods that access them directly. To detect all the mentioned features, he superimposes the *method call-graph* onto the concept lattice and obtains a *embedded call-graph*, which provides a detailed visualization of the class.

Godin and Mili [13] have applied concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. In their approach, they show how Cook's [6] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. They suggest how the design of a class hierarchy implementing the detected interfaces could be organized in a way that optimizes the distribution of the methods over the hierarchy. In C++ and Java, Snelting and Tip [24] have analysed a class hierarchy by making the relationship between methods and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. As a result, they propose a new class hierarchy that is behaviourally equivalent to the original one. Similarly, Huchard [14] and Leblanc [19] have applied concept analysis to improve the generalization/specialization of classes in a hierarchy.

There is also some relevant work to support the understanding of object-oriented systems at the class level that is not based on CA. GraphTrace visualizes concurrent animated views to understand the way a system behaves [15]. ObjectExplorer [17] uses both dynamic and static information that the reengineer can query and visualize via simple graphs to understand and verify hypotheses. Using basic graph visualizations to represent various relationships, Mendelzon and Sametinger [23] show that they can express metrics, verify constraints, and identify design patterns. Cross *et al.*, in the context of procedural languages, have proposes and investigated new control structure diagrams to support the reading of the applications' control flow [8]. Lanza and Ducasse have proposed *class blueprints*, which are structured call flows enriched with semantical information and metrics [18]. Finally, program slicing [11] is also used to support the understanding of programs. Based on slices, CodeSurfer [2] supports understanding by using hypertext facilities.

6 Conclusions and Future Work

In this paper we have applied concept analysis to help in the understanding of object-oriented classes. The identified concepts are the collaborations between groups of methods and attributes of a single class. Using them, we have defined a number of useful XRay views which correspond to groups of collaborations that expose specific aspects of a class, and they are particularly useful for understanding the behaviour of a class. We have validated the technique by applying it to a number of Smalltalk classes using ConAn, a tool we have developed to automatically generate collaborations that compose the XRay views.

In our first experiences we can observe the following:

- each XRay view has a clear focus, and identifies a set of methods exhibiting some key properties
- the views do not stand on their own, but complement and reinforce each other
- although the generation of collaborations and the views is fully automatic, their interpretation entails iterative application of views and opportunistic code reading
- the current approach does not take inheritance into account, which can be an impediment to understanding

Our next steps are to explore the definitions of new kinds of views, and apply them to larger classes. We also intend to explore ways of analysing classes in the context of their class hierarchies, and also considering the possible relationships with other class -not necessarily presented in the class hierarchies.

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). We thank Michele Lanza for his reviews.

References

[1] S. R. Alpert, K. Brown, and B. Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[2] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Proceedings of WISE'01 (International Workshop on Inspection in Software Engineering)*, 2001.

[3] V. Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[4] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[5] E. Casais and A. Taivalsaari. Object-oriented software evolution and re-engineering (special issue). *Theory and Practice of Object Systems (TAPOS)*, 3(4):233–301, 1997.

[6] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 1–15, Oct. 1992.

[7] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[8] J. H. Cross II, S. Maghsoodloo, and D. Hendrix. Control structure diagrams: Overview and evaluation. *Journal of Empirical Software Engineering*, 3(2):131–158, 1998.

[9] U. Dekel. Applications of concept lattices to code inspection and review. Technical report, Department of Computer Science, Technion, 2002.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *Transactions on Software Engineering*, 17(18):751–761, August 1991.

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

[13] 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.

[14] 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.

[15] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88*, volume 23, pages 191–205, Nov. 1988.

[16] S. Kuznetsov and S. Obëdkov. Comparing performance of algorithms for generating concept lattices. In *Proc. Int. Workshop on Concept Lattices-based KDD*, 2001.

[17] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA '95*, pages 342–357. ACM Press, 1995.

[18] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002*, pages 135–149, 2002.

[19] H. Leblanc. *Sous-hiérarchies de Galois: un modèle pour la construction et l'évolution des hiérarchies d'objets (Galois sub-hierarchies: a model for construction and evolution of object hierarchies)*. PhD thesis, Université Montpellier 2, 2000.

[20] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[21] C. Lindig. Fast concept analysis. In G. Stumme, editor, *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161. Shaker Verlag, 2000. ISBN: 3-8265-7669-1.

[22] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98, 1996.

[23] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16:170–182, 1995.

[24] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[25] Cincom VisualWorks Smalltalk. http://www.cincom.com/scripts/smalltalk.dll/.

[26] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[27] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, 83:445–470, September 1981.