XRay Views: Understanding the Internals of Classes

Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz

XRay Views: Understanding the Internals of Classes

Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz
Software Composition Group
University of 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 on the Smalltalk class* UIBuilder.

Keywords: Class Understanding, Concept Analysis, Logical Views

1. Introduction

Understanding source code is a key activity in the maintenance of software systems [3].

In the specific case of object oriented systems, reading the code is harder than procedural systems [4, 10], and therefore the maintenance is actually higher. This is due to several reasons [5]. The first issue is that, contrary to procedural languages, the method definition order in a file is not important [4]. 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 [5]. Another important problem is the presence of late-binding leads to "yoyo effects" when walking through a hierarchy and trying to follow the call-flow [13].

Focusing on classes, considered as cornerstones of object oriented systems, 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 to understand how a class works, we provide logically connected "XRay views" of classes that give the engineer an impression of the relationships between methods, attributes, and the invocation and access patterns of a class. In this way we support *opportunistic* understanding [11] in which the engineer understands a class iteratively by exploring patterns (given by the views) and reading code.

Taking into account the class as a sole unit, we are able to provide answers to the following questions about a class: (a) which methods access any attribute, directly or indirectly, (b) which groups of methods access directly or indirectly all the attributes or some subset of the attributes, (c) which methods are only called internally, (d) which methods/attributes are heavily used and accessed, (e) how the methods and attributes collaborate. Each of these aspects is important for understanding the inner workings of a class, but unfortunately they are dispersed 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. Concept analysis (CA) [6] is a branch of lattice theory that allows us to identify meaningful groupings of "objects" that have common "attributes" [1]. These groupings (known as *concepts*) form a partial order known as *concept lattice*. There are several algorithms for computing the concepts and the concept lattice for a given context [9]. For more details, the interested reader should refer to Ganter and Wille [6]. In this paper we apply this technique and limit our approach to understanding a single class, without taking into account relationships to subclasses, superclasses, or peer classes.

This paper is a short version of the approach introduced in [2], and is structured as follows: Section 2 introduces the definition of elements and properties used in ConAn, and the collaborations defined to build the XRay views. Sec-

1 To avoid confusion with object-oriented terminology, we refer in this paper instead to *elements* having common *properties*

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

tion 3 introduces in detail one specific view and a validation in the Smalltalk class UIBuilder. Sections 4 and 5 summarizes briefly the related work, our main conclusions and future work.

2. Applying Concept Analysis to Class Understanding

Complex software systems are composed of entities, such as classes, methods, modules, and subsystems, and 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. Note that we use the term *collaboration* to express a relationship between a set of methods and a set of attributes.

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 $m \in \mathcal{M}$ directly reads the value of attribute $x \in \mathcal{A}$
- m *writes* x means that the method $m \in \mathcal{M}$ directly updates the value of attribute $x \in \mathcal{A}$
- m *calls* n means that the method m calls the method n explicitly via a *self-call*.

We also define a number of derived properties, e.g.:

- 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})

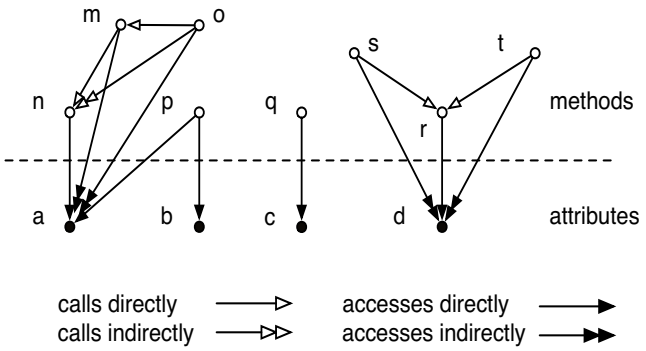

Figure 1. Attributes accesses and methods invocations and the groups they form

- 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 some concepts, e.g. $(\{m, o\}, \{\textit{accesses}^* \ a\})$, $(\{p\}, \{\textit{accesses} \ a, b\})$

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, $\forall e \in E, e' \in F, e' \ R \ e \Rightarrow e \in G$.

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. We will only list those that will be used later in the paper. The complete list of collaborations is listed in [2].

Direct Accessors: Direct accessors, readers or writers $M \subseteq \mathcal{M}$ of an attribute a are defined by a non-exclusive relationship: M *accesses* $\{a\}$. 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 express it as: M $\overline{accesses}$ $\{a\}$. In our example, we see that $\{r\}$ $\overline{accesses}$ $\{d\}$.

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 $\overline{accesses^*}$ $\{a\}$. 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 $\{s,t\}$ $\overline{accesses^*}$ $\{d\}$.

Collaborating Attributes: This collaboration expresses which attributes are used exclusively by a set of methods, and we express it as: M $\overline{accesses}$ A. In the example, we have the sets of attributes accessed exclusively by sets of methods are all of size 1: $\{q\}$ $\overline{reads}$ $\{c\}$ and $\{r\}$ $\overline{accesses}$ $\{d\}$.

Stateful Core Methods: This collaboration is a special case of *collaborating attributes* and expresses which methods access *all* the state of a class. We express it as: M $\overline{accesses}$ $\mathcal{A}$. 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.

3. XRay Views

An XRay view is a *group* of collaborations that exposes specific aspects of a class. Based on the collaborations specified above, we have defined 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. Because of the limited space of the paper, we provide a small explanation of all of them, but we only detail the definitions and case study of the view STATE USAGE. The three views are explained in detail in [2]. 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 the analysis of the class UIBuilder of the VisualWorks framework. It is a complex class that is used to build user interfaces (windows and their subcomponents) according to declarative specifications provided by its clients. We chose this class because it is complex enough in terms of number of instance variables (18) and methods (122) and communication between their methods, and it helps us to show characteristic results of XRay view application. As its name shows it is a Builder Design Patterns [1].

For the view, we ran our analysis tool, ConAn, on the chosen class, we examined the resulting view 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.

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 UIBuilder: Firstly, 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. 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.

4. Related Work

Within the CA application to understand software systems, we find several approaches. Dekel uses CA to visualize the structure of classes and to select an effective order for reading the methods [4]. Godin and Mili [7] applied concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. In C++ and Java, Snelting and Tip [12] analysed a class hierarchy by making the relationship between methods and variables explicit. Similarly, Huchard [8] applied concept analysis to improve the generalization/specialization of classes in a hierarchy.

5. 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. We only introduced the view STATE USAGE which helps to understand how the state of the class is used. Two other views are defined and introduced in [2]. Each of them expose specific aspects of a class in terms of groups of collaborations. We have limited our validation in this paper to the Smalltalk class UIBuilder, but also the classes Scanner and OrderedCollection were analyzed in [2]. In our validadtion, we use 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 a limitation 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 and collaborations 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] G. Arévalo. Understanding classes using x-ray views. In *Proceedings of 2nd. MASPEGHI (ASE 2003)*, 2003.

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

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

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

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

[7] 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.

[8] 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.

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

[10] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[11] 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.

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

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