X-Ray Views on a Class using Concept Analysis [*]

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

Abstract

Within object oriented software, the minimal unit of development and testing is a class. So understanding how a class is defined and behaves is important. Considering that a class is composed of instance variables and methods, the process is not so easy to achieve because we must decide which different viewpoints can help us to detect features of a class. These viewpoints can include identifying groups of methods accessing a (set of) instance variable(s), groups of methods that interact among themselves to provide a functionality or groups of methods that behave as interface. Thus, with these different groups, we are able to know the different hidden characteristics of a class. In this position paper, we propose to apply Concept Analysis to generate the different groups of (collaborating) entities and use these groups to define different views. These views will help us to get the main features of a class.

1. Introduction

Analyzing or reusing an object oriented system implies understanding how the different classes are built and working, and how the classes interact together to provide the different functionalities of the system. The functionality of a class is defined in terms of its state (instance variables) and behavior (methods). Thus the way a class is working depends on the dependencies between these entities. In this position paper, we propose to identify these dependencies grouped in **views** and characterize the class based on them. We propose to identify the dependencies between entities using Concept Analysis, and to provide an interpretation of the views to characterize a class. This paper is structured as follows: First, we introduce which are the different aspects we can take into account to understand a class. Second, we introduce our approach defining briefly what is Concept Analysis and how we use it in our study context. Third, we define several dependencies and we group them to build the views. Finally, we show the validation on the *Smalltaltk* class OrderedCollection and the conclusions we get from this experiment.

[*]In Proceedings of WOOR 03 (4th International Workshop on Object Oriented Reengineering) colocated in ECOOP 2003, pp. 76-80, University of Antwerp, 2003

2. Class Understanding

Within object oriented software, the minimal unit of development and testing is a *class*. Usually, a class is composed of *instance variables* used to represent the state, and *methods* used to represent the behaviour. Then, understanding how a class works means identifying several aspects [2, 3]:

- how the methods are interacting together (coupling between methods)
- how the instance variables are working (or not) together in the methods (coupling between instance variables)
- which methods are using (or not) the state of the class
- if there are methods that form a cluster and define together a precise behaviour of the class
- which methods are considered as interfaces
- which methods are used as entry points (methods that are considered as interfaces and communicate with other methods defined in the class)
- which methods and instance variables represent the core of the class
- which methods are using all the state of the class

These aspects are not explicit and not simple to detect and manage. Identifying clear criteria to classify the different *dependencies* among the instance variables or methods allows to get a characterization of the class and understand how the class is working and collaborating with other classes in a system. We call **view** the group of dependencies.

3. Our approach: Definition of Views

Our approach is based on two main aspects: *identification and grouping of dependencies in* ***views*** in the class and *interpretation of the* ***views*** applied in a class. The following sections explain these different aspects. First we explain briefly the main concepts of *Concept Analysis*, which is the technique we use to identify the different dependencies, and how we apply it in our approach. Secondly, we give a notation for these dependencies to avoid any case of ambiguity. Following we propose 2 **views** based on grouping of dependencies and an interpretation of them applied on a class. Finally, as a validation we show how we can apply the views on the *Smalltalk* class OrderedCollection

4. Concept Analysis

Concept Analysis (CA) is a branch of lattice theory that allows us to identify meaningful groupings of *elements* (referred to as *objects* in CA literature) that have common *properties* (referred to as *attributes* in CA literature) [1]. These groupings are called *concepts* and capture similarities among a set of *elements* based on their common *properties*. Mathematically, concepts are *maximal collections of elements sharing common properties*. They form a complete partial order, called a *concept lattice*, which represents the relationships between all the concepts [1, 5, 4]. To use the CA technique, one only needs to specify the properties of interest on each element, and does not need to think about all possible combination of these properties, since these groupings are made automatically by the CA algorithm. The possibility of capturing similarities of elements in groups (*concepts*) -based on the specification of simple properties- allows to identify common features of the elements. In the specific case of software reengineering, the system are composed of a big amount of different entities (classes, methods, modules, subsystems) and there are different kinds of relationships among them. When we are able to characterize the entities in terms of properties, and we can detect if these characteristics are repeated in the system, then we can reduce the amount of information to analyze and we can have an abstraction of the different parts of a system. These abstractions helps us to start to see how the parts are working, how they are defined and how they are connected to others parts of the system. In our specific case, we are interested in seeing how the class is structured internally in terms of instance variables and methods. Then, CA will give us groups *(concepts)* that allows us to analyze different dependencies inside the class.

[1]We prefer to use the terms *element* and *property* instead of *object* and *attribute* because the latter terms have a specific meaning in the object-oriented paradigm.

4.1. Use of Concept Analysis: Definition of Elements and Properties

First we need to define the elements and the properties that are used to calculate the *concepts* by the CA algorithm. In the specific case of our case study, we are focused on the analysis of the class as a sole unit, thus the elements are the instance variables and the methods defined in the class, and the properties are how they are related between themselves. Consider that we have the set of instance variables {*A*, *B*}, and the set of methods {*P*, *Q*, *X*, *Y*} defined in a class, the properties we use are:

- *B is used by P* means that the method *P* is accessing directly or through an accessor/mutator to the instance variable *B*.
- *Q is called in P* means that the method *Q* is called in the method *P* via a *self-call*

The predicates presented previously represent direct *dependencies* between the involved entities. But the interesting/curious point is to consider an *indirect dependency* between two entities, for example the instance variable *B is read by X* and that *X is called in P*. If we consider this case, we say that obviously *B is read by X* but also indirectly *B is read by P* too. With these kinds of dependencies we will be able to identify not only groups of direct accesses/uses of entities, but also indirect ones that will help us to measure the impact of the changes and the level of dependency between instance variables and methods in a class.

4.2. Notation

We introduce a notation to define the dependencies mentioned previously without ambiguity. As we said previously, these dependencies are the contents of the concepts calculated by the Concept Analysis algorithm.

- $\{E_1, ..,E_n\}\ \overline{R}\ \{M_1, ..,M_p\}$ means that the entities $\{E_1, .., E_n\}$ *depend exclusively* on $\{M_1, .., M_p\}$. This means that $\{M_1, .., M_p\}$ are the only entities that are related through the property *R* to $\{E_1, ..,E_n\}$.
- $\{E_1, ..,E_n\}\ R\ \{M_1, ..,M_p\}$ means that the entities $\{E_1, ..,E_n\}$ do *not depend exclusively* on $\{M_1, ..,M_p\}$.

This means that $\{M_1, .., M_p\}$ is a subset of all the entities that are related through the property *R* to $\{E_1, .., E_n\}$.

- $\{E_1, .., E_n\}$ $\overline{R}^*$ $\{M_1, .., M_p\}$ means that the entities $\{E_1, .., E_n\}$ *depend exclusively and transitively* on $\{M_1, .., M_p\}$. This means that $\{M_1, .., M_p\}$ are the only ones that are related to $\{E_1, .., E_n\}$ through the property *R* and R_1, where R_1 is an intermediate property, because there is a set $\{N_1, .., N_k\}$ such that: $\{E_1, .., E_n\}$ $\overline{R}$ $\{N_1, .., N_k\}$ $\overline{R_1}$ $\{M_1, .., M_p\}$
- $\{E_1, .., E_n\}$ R^* $\{M_1, .., M_p\}$ means that the entities $\{E_1, .., E_n\}$ do *not depend exclusively but transitively* on $\{M_1, .., M_p\}$. This means that $\{M_1, .., M_p\}$ are not the only ones that are related to $\{E_1, .., E_n\}$ through the property *R* and R_1, where R_1 is an intermediate property, because there is a set $\{N_1, .., N_k\}$ such that: $\{E_1, .., E_n\}$ *R* $\{N_1, .., N_k\}$ R_1 $\{M_1, .., M_p\}$
- A special case: $\{E_1, .., E_n\}$ $\neg\overline{R}$ $\{M_1, .., M_p\}$ means that the entity $\{E_1, .., E_n\}$ *has any dependencies* on $\{M_1, .., M_p\}$. This is only applicable on *exclusive dependencies*.

4.3. Dependencies: Definition and Interpretation

Based on the proposed notation, we define the different dependencies we can have in a class:

Direct "Accessors": We consider a method as *direct accessor* when the method uses a specific instance variables directly or through a *self-call* to the method accessor or mutator of the instance variable. And it is represented with a non-exclusive dependency as:

- $\{I_j\}$ *is used by* $\{M_1, .., M_k\}$

This dependency allows us a simple classification of the methods according which instance variables use. As it is a non-exclusive dependency, if there is a method that uses several instance variables, this method will be in the set of methods of each instance variable where it has influence.

Exclusive Direct "Accessors": Similar to the previous defined dependency, but we consider a method as *exclusive direct accessor"* when the method uses only a specific instance variable directly or through a *self-call* to the method accessor or mutator of the instance variable. As it is represented with a exclusive dependency as:

- $\{I_j\}$ $\overline{is\ used\ by}$ $\{M_1, .., M_k\}$

Similarly to the previous dependency, we classify the methods according which instance variables use. But in this case, we consider exclusive dependency, this means that the groups of methods are disjoint among them. If there is a method that access several instance variables, it will not appear in this dependency.

Exclusive Indirect "Accessors": We consider a method as *exclusive indirect accessor* when the method calls a *direct accessor* method of a specific instance variable. It is represented with an exclusive dependency as:

- $\{I_j\}\overline{is\ used\ by}\{M_{k+1}, ...M_p\}$
  if $\{I_j\}is\ used\ by\{M_1, ...M_k\}is\ called\ in\{M_{k+1}, ...M_p\}$

This dependency helps us to distinguish those methods that define the behavior of a class without using at all the state from those that use the state of the class.

Collaborating instance variables: This dependency expresses which instance variables are being used at the same time in a (set of) method(s). It is represented with an *exclusive dependency* as:

- $\{I_1, .., I_j\}\overline{is\ used\ by}^*\{M_1, .., M_k\}$

Although the definition seems to be similar to the previous ones, the main difference is that in this case, we are focusing on having set of instance variables working together. This dependency helps to identify potentially which are the functionalities that sets of instance variables are providing inside the class. We are not limited to methods that access directly to the instance variables, but we are focused on identifying all the methods that can affect the state of a class.

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

- $\{M_1, .., M_j\}\neg\overline{is\ called\ in}^*\{M_{j+1}, .., M_k\}$

This dependency helps us to define which are the methods that are used as interface to the class. We consider that if they are not used inside the class, these methods are called outside the scope of the class.

Internal Behavior: This dependency expresses which methods are used inside the class. It is represented with an *exclusive dependency* as:

- $\{M_1, .., M_j\}\overline{is\ called\ in}^*\{M_{j+1}, .., M_k\}$
  $if\{I_1, .., I_j\}\neg is\ used\ by\{M_1, .., M_j\}$

Clearly, this dependency is complementary to the *Interface methods*, and helps to identify methods that are defining internally the behaviour of the class but this behaviour is not related to modifying/accessing the state of the class.

Externally Used State : This dependency expresses which methods are *exclusive direct accessors* and *interface methods*. It is represented with an *exclusive dependency* as:

- $\{M_1, .., M_j\} \neg \overline{is\ called\ in}^* \{M_{j+1}, .., M_k\}$ if $\{I_j\} \overline{is\ used\ by}^* \{M_1, .., M_j\}$

This dependency helps us to define which are the methods that are used as interface to the class and accessing directly to the state of the class.

5. Views and Validation

Based on the dependencies defined previously, we introduce two views that can be applied in a class: *Core Attributes* and *Public Interface*. They are simple but show how useful the combination of *dependencies* is.

We show a validation on the *Smalltalk* class OrderedCollection. It 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 are indices. This class has two instance variables {*firstIndex, lastIndex*} to manage the indexes of the first and last elements in the collection. Its behavior is defined in 56 methods.

5.1. View: Core Attributes

Description: Cluster methods according to their access to instance variables

Used dependencies:

- exclusive direct accessors,
- exclusive indirect accessors, and
- collaborating instance variables

Rationale: Instance variables represent the state of an object. Understanding how the state of an object is accessed provides key information regarding the class structure and behavior. In particular the distribution of the groups regarding the number of instance variables helps understanding how the class is internally built.

Validation on OrderedCollection: When we identify the dependencies, we have:

- {firstIndex} $\overline{is\ used\ by}${before, removeAtIndex:, addFirst:, add:beforeIndex:, removeFirst, removeFirst:, first}
- {lastIndex} $\overline{is\ used\ by}${removeIndex:, after:, addLastNoCheck:, removeLast, addLast:, removeLast:, last, }
- {firstIndex, lastIndex} $\overline{is\ used\ by}^1$ {makeRoomAtFirst, changeSizeTo:, removeAllSuchThat:, makeRoomAtLast, do:, notEmpty:, keysAndValuesDo:, detect:ifNone:, changeCapacityTo:, isEmpty, size, remove:ifAbsent:, includes:, reverseDo:, find:, setIndices, insert: before:, at:, at:put:, setIndicesFrom:}
- {firstIndex} $\overline{is\ used\ by}^*$ {addAllFirst:}
- {lastIndex} $\overline{is\ used\ by}^*$ {addAllLast:, add:}.
- {firstIndex, lastIndex} $\overline{is\ used\ by}^*$ {add:before:, add:after:}

Discussion: According to these results we see that most of the methods access directly to the state of the class, and only a few ones are accessing indirectly to the class. This means that the behavior of the class is not used internally, and most of it is directly in terms of the instance variables. In this particular case, we see that the core attributes {*firstIndex, lastIndex*} have no *pure accessors* or *mutators*. This means that the state is used only internally and it is not exported at all outside the class.

5.2. View: Public Interface

Description: Identify methods that behave as the public interface of the class

Used dependencies:

- interface methods
- externally used state

Rationale: Methods define the internal and external behavior of a class. Classifying them in terms of this feature helps us to identify which is the *interface* for the class

Validation on OrderedCollection: In Ordered Collection, as we saw in the previous view that most of the methods are behaving as interface methods.

Discussion: In this particular case we see that it almost does not exist methods focused on internal functionalities of the class, and most of the methods are exported to be used as interfaces.

6. Conclusions and Future Work

In this position paper, we propose a technique to understand how a class itself is defined and working. Thus, we identify known (e.g., *instance variable A is used by method M*) but also unknown dependencies (e.g., *instance variable A is used indirectly by method M*) between instance variables and methods using Concept Analysis. Afterwards, we group these dependencies in **views** and we give an interpretation of the meaning of these dependencies when we apply them to a class. We provide a validation in the *Smalltalk class OrderedCollection*. In this particular case we see that there is no internal defined functionality for the class and that most of the methods are behaving as *interface*. We also see that the only two instance variables (complete state) of the class are being accessed at the same time by several methods, meaning that there is good coupling between them.
We believe that these views show partially how the class is working but they are useful because they are reducing the amount of information provided by the class itself, and help the developer to have different *viewpoints*.
As we are using Concept Analysis -as our *generator of information*-, it is important to remark that the groups we use in the *views* are automatically generated by the concept analysis algorithm. We only propose a possible interpretation to the generated groups based on the combination of the properties given by the algorithm.
Another point to take into account is that we use *simple* properties to express the dependencies between instance variables and methods. But the combination of the information of the groups (provided by the lattice) makes us understand *complex* dependencies inside the class.
Next steps include the definition of new views and the validation of them in large classes, and also analyse the impact of superclass and subclasses in this approach.

Acknowledgements: 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)

References

[1] G. Birkhoff. Lattice theory. *American Mathematical Society*, 1940.
[2] E. Casais. An incremental class reorganization approach. In O. L. Madsen, editor, *Proc. ECOOP ’92*, volume 615 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
[3] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
[4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
[5] 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.