

# **Revealing JAVA Class Structure with Concept Lattices**

Research Thesis

Submitted in partial fulfillment of the requirements for  
the degree of Master of Science in Computer Science

**Uri Dekel**

Submitted to the Senate of  
the Technion — Israel Institute of Technology

Adar I 5763

Haifa

February 2003

This research was done under the supervision of Dr. Yossi Gil in the department of Computer Science at the Technion. I would like to thank Yossi especially for his instructive guidance in scientific writing.

This work is dedicated to the loving memory of my late grandparents, Arie and Bracha Levy, who always supported me and believed in me.

Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence.

# Preface

The notion of a class in object oriented programming is a natural candidate for reuse. Unfortunately, programmers tend to be wary of trusting classes written by other developers, and often prefer to reinvent the wheel. The reasons for this mistrust are justified in many cases, even in class libraries aimed for general reuse: Many classes suffer from problematic encapsulation, sensitivity to change, inconsistencies in their interfaces, and poor documentation. These problems significantly increase the effort required for the integration of a third-party class into an existing system, making reuse impractical.

The popularity of the JAVA language, and especially its inherent support for supplying users with compiled classes for all platforms without requiring recompilation or risking header inconsistencies, opens many new opportunities for the reuse of classes. Yet, if we want programmers to take advantage of these opportunities, we need to make sure that these classes are simple to understand and are consistent in their features and implementation. It is difficult to assure these properties because some classes are very large, often spanning dozens or even hundreds of methods.

The process by which a human learns how to use a class is cognitive and cannot be automated. Assuring the quality of a class involves some tasks that can be automated, but many others depend on manual techniques such as code inspection. Yet, the human effort involved with these tasks can be greatly reduced with appropriate tools and methodologies, such as software visualization tools.

Our work suggests, for the first time, that the technique of formal concept analysis (FCA) be applied to individual classes. We use this technique in an attempt to automatically classify the features of these classes, facilitating their reuse, and to reveal their internal structure, for purposes of quality assurance or reverse engineering. FCA is a mathematical classification technique which was previously applied to different problems in software research, such as modularizing legacy code and restructuring class hierarchies.

In this work, a formal concept consists of a pair of maximal sets, one of methods and one of fields, such that each field is used by every method and each method uses every field. The partial order between concepts is defined by the inclusion relation between their respective sets, and visualized in the form of a concept lattice. We argue that the set of fields representing a class is less volatile than the set of services it provides, and that in most cases, all possible implementations of the same operation will use the same fields. Therefore, usage patterns of fields by methods are fundamental to understanding the functionality and the implementation of a class, and can serve as a heuristic for classifying similar methods and revealing inconsistencies.

After the access relation between methods and fields is automatically elicited from a Java class file, a class concept lattice is drawn. The user can then employ a variety of tools to abstract the class information (zooming out), and focus on interesting details (zooming in). These tools are discussed as part of a semi-structured 3-stage methodology of applying FCA to the exploration of a JAVA class: learning its interface, examining the implementation, and inspecting its code.

Two real-life case studies demonstrate the methodology and the various zoom-in and zoom-out views of the concept lattice. Initial evidence to the efficacy of the methodology is that we revealed problems which were confirmed as previously-unknown errors and are expected to be fixed in future versions. Preliminary user studies show that developers with no previous familiarity with the technique were able to use it to discover problems in actual classes following very short tutorials. A larger

and more systematic user study is needed to fully validate the methodology.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Definitions of Accesses Between Fields and Methods</b>	<b>5</b>
2.1	The set of fields of a class . . . . .	5
2.2	The set of methods of a class . . . . .	7
2.3	The relation of fields accessed by methods . . . . .	8
2.4	Approximating the field accesses relation . . . . .	9
2.4.1	Simplifications . . . . .	10
2.4.2	Calculation process . . . . .	12
<b>3</b>	<b>Review of Formal Concept Analysis</b>	<b>16</b>
3.1	Formal contexts . . . . .	16
3.2	Formal concepts . . . . .	17
3.2.1	Galois Connection . . . . .	17
3.2.2	Concepts . . . . .	18
3.2.3	Order between concepts . . . . .	20
3.3	Concept lattices . . . . .	22
3.3.1	Sparse annotation . . . . .	23
3.3.2	An algorithm for computing concepts and constructing lattices . . . . .	24
<b>4</b>	<b>Concept Lattices of Classes</b>	<b>27</b>
4.1	Interpretation of concept lattices . . . . .	27
4.2	Additional contexts . . . . .	28
4.2.1	Subcontexts . . . . .	28
4.2.2	Selecting Methods . . . . .	29
4.2.3	Selecting Fields . . . . .	32
4.2.4	Selecting Accesses . . . . .	33
<b>5</b>	<b>Methodology Stage I: Interface Analysis</b>	<b>37</b>
5.1	Introduction . . . . .	37
5.2	The CDK case study . . . . .	38
5.3	Step 1: Set expectations . . . . .	38
5.4	Step 2: Explore the environment of the class . . . . .	41
5.5	Step 3: Select a context . . . . .	43

5.6	Step 4: Lay-out the concept lattice using layers . . . . .	44
5.7	Step 5: Simplify concepts' annotations . . . . .	46
5.8	Step 6: Perform horizontal decomposition . . . . .	48
5.9	Step 7: Create an abstraction lattice . . . . .	50
5.10	Step 8: Match services against expectations . . . . .	52
5.11	Step 9: Identify core, auxiliary and wrapper services . . . . .	54
<b>6</b>	<b>Methodology Stage II: Implementation Analysis</b>	<b>55</b>
6.1	Introduction . . . . .	55
6.2	Step 1: Construct the embedded call graph . . . . .	56
6.3	Step 2: Identify unused fields . . . . .	58
6.4	Step 3: Discover the roles of fields . . . . .	58
6.5	Step 4: Investigate interdependencies between fields . . . . .	59
6.6	Step 5: Assess the quality of the names of fields . . . . .	61
6.7	Step 6: Investigate methods which access the entire state . . . . .	61
6.8	Step 7: Investigate methods which do not access any part of the state . . . . .	62
6.9	Step 8: Study asymmetries . . . . .	64
6.10	Step 9: Study the access patterns of methods . . . . .	65
6.11	Step 10: Examine non-public methods . . . . .	65
6.12	A summary of the problems discovered in the <code>Molecule</code> class . . . . .	66
<b>7</b>	<b>Methodology Stage III: Code Inspection</b>	<b>69</b>
7.1	Introduction . . . . .	69
7.2	The difficulties of inspecting object oriented systems . . . . .	69
7.3	Reading order . . . . .	70
	7.3.1 Global order . . . . .	71
	7.3.2 Local order . . . . .	72
7.4	Inspection tasks . . . . .	73
<b>8</b>	<b>Case Study: Applying the methodology to the <code>Graph</code> class of <code>VGJ</code></b>	<b>75</b>
8.1	Introduction . . . . .	75
8.2	Stage I - Analyzing the interface . . . . .	76
	8.2.1 Preliminaries . . . . .	76
	8.2.2 Constructing and simplifying the lattice . . . . .	77
	8.2.3 Studying the lattices . . . . .	79
8.3	Stage II - Analyzing the implementation . . . . .	80
	8.3.1 Investigating fields . . . . .	80
	8.3.2 Investigating Methods . . . . .	82
8.4	Stage III - Inspecting the code . . . . .	83
<b>9</b>	<b>Conclusions and Future Research</b>	<b>87</b>
9.1	A lattices based metrics suite . . . . .	88
9.2	Interactive design of classes . . . . .	89

<b>A</b>	<b>Previous Applications of Concept Analysis in Software Engineering</b>	<b>90</b>
A.1	Modularization . . . . .	91
A.2	Class hierarchies . . . . .	93
A.3	Component retrieval . . . . .	94
A.4	Reverse engineering . . . . .	94
A.5	Program comprehension . . . . .	95

# List of Figures

1.1	Illustration of a concept lattice for a <code>Rectangle</code> class . . . . .	2
1.2	An FCA-based zoom-in zoom-out toolbox . . . . .	3
2.1	Source code of class <code>Pnt</code> . . . . .	6
2.2	Source code of class <code>Pnt3D</code> . . . . .	6
2.3	A class hierarchy where dynamic dispatching affects the accesses relation . . . . .	11
3.1	A scatter graph showing the number of concepts relative to $\min( \mathbf{O} ,  \mathbf{A} )$ in the JDK . . . . .	20
3.2	A scatter graph showing the number of concepts relative to $\min( \mathbf{O} ,  \mathbf{A} )$ in <i>Eclipse</i> . . . . .	21
3.3	A scatter graph showing the number of concepts relative to $ \mathbf{A} $ in the JDK . . . . .	21
3.4	A scatter graph showing the number of concepts relative to $ \mathbf{A} $ in <i>Eclipse</i> . . . . .	22
3.5	Fully annotated concept lattice of the concrete context of class <code>Pnt3D</code> . . . . .	23
3.6	Concept lattice of the concrete context of class <code>Pnt3D</code> , using sparse annotations. . . . .	24
3.7	Fully annotated concept lattice of the <code>Pnt</code> class. . . . .	26
4.1	Concept lattice of class <code>Pnt3D</code> after field <code>z</code> is removed from the context. . . . .	29
4.2	Concept lattice of all the accesses to fields in the <code>sun.net.www.MimeEntry</code> class. . . . .	34
4.3	Concept lattice of read accesses to fields in the <code>MimeEntry</code> class. . . . .	36
4.4	Concept lattice of write accesses to fields in the <code>MimeEntry</code> class. . . . .	36
5.1	Hierarchy of classes in the root package of CDK . . . . .	41
5.2	Summary of context selection decisions . . . . .	44
5.3	Layers in an example lattice . . . . .	45
5.4	Concept lattice of <code>Molecule</code> with full signatures and no fields . . . . .	45
5.5	Summary concept lattice of <code>Molecule</code> with no fields . . . . .	47
5.6	Outline concept lattice of <code>Molecule</code> with no fields . . . . .	48
5.7	Horizontal decomposition of an example lattice . . . . .	49
5.8	Horizontal decomposition of the outline concept lattice of <code>Molecule</code> . . . . .	49
5.9	Abstraction lattice of component $L$ of <code>Molecule</code> superimposed on the original lattice. . . . .	52
5.10	Concepts in the <code>bonds</code> and <code>add_bonds</code> clusters of <code>Molecule</code> . . . . .	53
6.1	Embedded call graph of class <code>Pnt3D</code> . . . . .	56
6.2	Embedded call graph of component $L$ in class <code>Molecule</code> . . . . .	57
6.3	Trivial components in the lattice of <code>Molecule</code> . . . . .	58
6.4	A zoomed-in concept lattice of component $L$ of <code>Molecule</code> with fields. . . . .	59
6.5	Implications between fields in class <code>Graph</code> . . . . .	60

6.6	Concept lattice of <code>Molecule</code> annotated with the classes that define each member. .	62
6.7	A concept lattice of <code>Molecule</code> with methods at all access-levels . . . . .	66
8.1	Concept lattice of <code>Graph</code> with full signatures and no fields . . . . .	77
8.2	Summary concept lattice of <code>Graph</code> . . . . .	78
8.3	Outline concept lattice of <code>Graph</code> . . . . .	78
8.4	Abstraction lattice of <code>Graph</code> . . . . .	78
8.5	Concept lattice of <code>Graph</code> with non-public members . . . . .	81
8.6	Embedded call graph of <code>Graph</code> restricted to the top concept . . . . .	82
8.7	Embedded call graph of <code>Graph</code> . . . . .	84

# List of Tables

2.1	<i>Direct field-accesses</i> relation of the <code>Pnt3D</code> class . . . . .	8
2.2	<i>Field-accesses</i> relation of the <code>Pnt3D</code> class . . . . .	9
2.3	<i>Direct method calls</i> relation of the <code>Pnt3D</code> class . . . . .	13
2.4	<i>Method-calls</i> relation of the <code>Pnt3D</code> class . . . . .	14
3.1	<i>Complete context</i> of the <code>Pnt3D</code> class . . . . .	17
3.2	<i>Concrete context</i> of the <code>Pnt3D</code> class . . . . .	18
3.3	Concepts for the concrete context of the <code>Pnt3D</code> class . . . . .	19
3.4	<i>Complete context</i> of the <code>Pnt</code> class . . . . .	25
4.1	Method selection options . . . . .	30
4.2	Method selection options for the concrete context . . . . .	32
4.3	Field selection options . . . . .	33
4.4	Access types selection options . . . . .	33
4.5	Field accesses relation in the <code>sun.net.www.MimeEntry</code> class. . . . .	35
5.1	Basic <i>functionality annotations</i> for naming concept responsibilities . . . . .	47
5.2	Composite <i>functionality annotations</i> for naming concept responsibilities . . . . .	48
5.3	Abstraction context of component $L$ from the concept lattice of <code>Molecule</code> . . . . .	51
6.1	Symmetric concepts in the lattice of the <code>Molecule</code> class. . . . .	64
8.1	Metrics for the methods of concept 2 in the lattice of <code>Graph</code> . . . . .	86
A.1	Works on applications of concept analysis, by category . . . . .	91

# Chapter 1

## Introduction

One of the reasons for the success of the object oriented paradigm is the ability of classes to conveniently model program-space and problem-space entities. Yet, even though many of these entities recur across multiple applications, the majority of classes (aside from meticulously-developed library classes) are rarely reused.

Programmers tend to be wary of trusting classes written by other developers, and often prefer to “reinvent the wheel”. The reasons for this mistrust are often justified, even for classes that were developed with future reuse in mind. Many classes are simply too hard to understand and use. In some cases, they are not documented at all, or their documentation does not accurately describe the features and capabilities that they provide. In others, the mapping from the provided features into actual interface methods is unclear or unnatural due to a lack of a consistent naming scheme; if one has to look up a method name every time a certain feature is needed, the reuse effort may not be worthwhile.

In order to enhance the prospects for reuse, Meyer proposed the *shopping list approach* [42, pp.80–83]. This approach encourages class developers to provide a complete interface with *composite methods* for common user-operations, without adding functional power to the class. However, following this approach is a double-edged sword: while it removes duplicate expressions from user programs, it can significantly bloat the interface of the class. Potential customers may be apprehensive of a large interface with dozens of methods and deem the effort of reusing the class not worthwhile.

It is difficult to understand how a class is used or to locate a specific functionality when one is confronted with a long list of methods, such as that provided by the JAVADOC utility [29]. Means are needed to convey the same information in a more convenient and meaningful form, for example by rearranging the methods (e.g., [30]). Meyer suggests the process of *feature categorization* [42, pp.103–108], in which methods are arranged in functional groups. However, the availability of this categorization depends on the good will of the class suppliers, who rarely perform such an explicit organization. Furthermore, even if such an order existed when the class was first designed, Belady and Lehman’s *laws of program evolution dynamics* [3] suggest that the entropy of the class will increase as it evolves, and this may result in the loss of the indented categorization.

In this thesis we suggest a heuristic for performing an automatic feature categorization of the methods of a class. Our heuristic is the use of fields by methods: if two methods use the same set of fields, then they are placed in the same category. The underlying rationale is that the fields used by a method play an important role in the definition of its semantics, and therefore methods that are

related in their semantics will use the same fields. We base this argument on two postulates, that the structure of an object is less volatile than its interface, and that all the implementations of the same functionality will usually make use of the same fields; We believe that it is rare to see two equivalent yet different representations of an entity, such as the polar and cartesian representations of complex numbers.

One of the problems of using Meyer’s proposal of a linear list of orthogonal feature categories is that each category can be semantically connected to several others, or may semantically contain other categories. For example, consider a class that represents a rectangle and has four fields: `top`, `left`, `width` and `length`. Some of the methods of this class, such as the setters and getters of the individual fields, operate on a single field at a time. Others, such as translation and resizing, make use of several stage variables, and few, such as drawing the rectangle, will make use of the entire state of the object. We believe that the representation of such a class should account for the multiple connections between the categories and also for their hierarchical organization. Therefore, we propose that the class be presented in the form of a certain *Hassé diagram* called a *concept lattice*, as illustrated in Fig. 1.1. Each diagram node, called a concept, represents a set of methods that use the same set of fields: the fields introduced in that concept and in all the concepts below it.

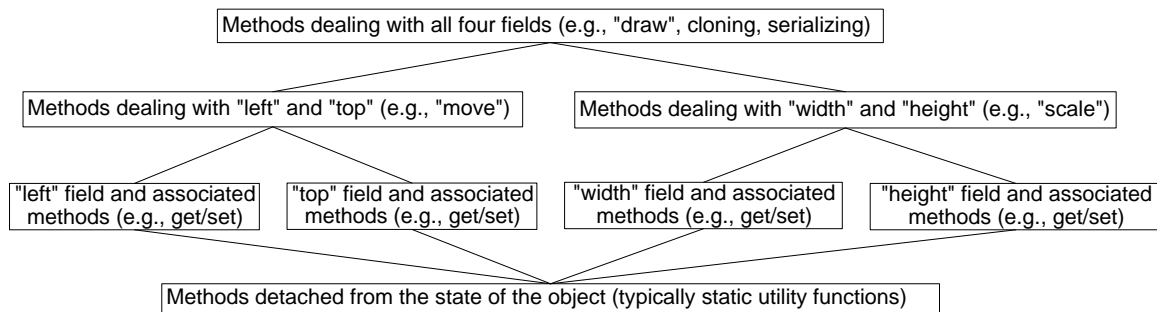


Figure 1.1: Illustration of a concept lattice for a `Rectangle` class

Concept lattices are the primary result of performing *formal concept analysis* (FCA) on a binary relation. FCA is a technique for clustering abstract entities, commonly called *objects*, that share common *attributes* into *formal concepts*. The technique, germinated by Birkhoff [8] and considerably enriched by Ganter and Wille [22, 60] found many different applications in software engineering. These applications include configuration management [49], management of software component libraries [20, 38, 47], construction and maintenance of class hierarchies [26, 27, 51], software modularization [48]) and other topics, discussed in Appendix A.

Note that while the concept lattice conveys exactly the same information as the binary relation for which it was constructed, it presents that information in a more meaningful way. Each concept represents an equivalence class inside the relation, and the lattice portrays the dominance relations between these classes.

In addition to improving the understandability of a class, we argue that the partitioning of methods by their use of fields and their arrangement in the form of a concept lattice, can help discover both interface and implementation problems in the class. Such problems include asymmetries and inconsistencies between interface methods, redundant methods, potential breaking of class invariants, incorrect implementation of copy operations, etc. These problems are not as easily discovered using

a traditional code inspection, because readers are distracted by style and implementation problems in the individual methods and fail to see the bigger picture. Preliminary user studies that we conducted appear to confirm that many of these problems are more likely to be discovered when a concept lattice is used instead of the source code. This was true even when the participants had no familiarity with the JAVA language (but some C++ knowledge) and were given a very short explanation about how concept lattices are interpreted.

This thesis presents an elaborate three-stage methodology for the analysis and inspection of an isolated JAVA class. In the first stage, the public interface of the class is analyzed and understood. Hidden fields and other implementation details are examined in the second stage, where additional problems are discovered. The actual source code of the class, if available, is only consulted in the third stage, where the concept lattice is used to select a reading order that may improve the effectiveness of the code inspection.

We chose to focus on the analysis of JAVA classes for two reasons. First, the popularity of the language and its inherent properties, including its security features and its multiplatform binary class format, make it an interesting case study language for class reuse. We believe that since JAVA addresses many problems that prevented reuse in older languages, such as C++, most of the reasons for avoiding reuse in JAVA are due to the effort required for understanding and using the class rather than due to integration problems. Second, the binary class-file format of JAVA allows us to easily extract the information required for the construction of the concept lattice, even if the original source code is absent; this makes our methodology useful for the reverse engineering of classes. In older languages, the source code is usually required to obtain the same information. Nevertheless, we believe that much of our proposed methodology can be applied to other languages, and leave the details to future research.

In presenting the methodology, we rely on a toolbox of *zoom-in* and *zoom-out* tools (see Fig. 1.2) for elaborating and simplifying the lattice according to the exact needs. The effect of applying a tool can be limited to the annotations of the concepts themselves (e.g., the removal of signatures or the naming of concepts), but it can also change the actual structure of the lattice (e.g., by clustering concepts), or add new information, such as details from the call-graph.

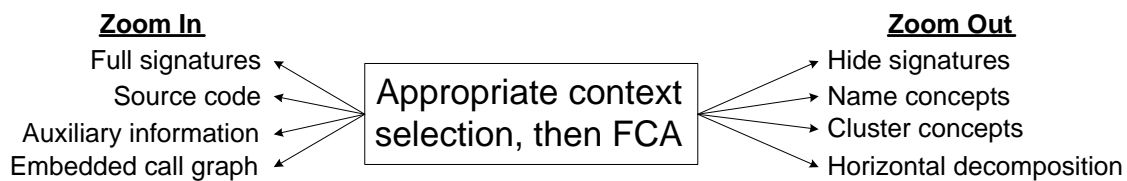


Figure 1.2: An FCA-based zoom-in zoom-out toolbox

The methodology and the use of the toolbox are demonstrated in two case studies of real-life open-source programs. Initial evidence to the efficacy of the methodology is that we revealed problems which were confirmed as previously-unknown errors.

We have a prototype tool for computing and drawing the concept lattice of a JAVA class, and we are currently working on an interactive tool which fits into the *Eclipse* development framework [15]. In addition, we plan a more systematic user study to fine-tune and validate the methodology.

## Outline

This thesis is structured as follows: Chapter 2 defines the sets of methods and fields of a JAVA class and the relation of accesses between them. The process of performing a *formal concept analysis* of this relation is described in Chapter 3. Suggestions for interpreting the resulting concept lattice are provided in Chapter 4, which also presents various options for simplifying the underlying accesses relation. The three stages of our class analysis methodology: interface analysis, implementation analysis, and code inspection are discussed in chapters 5, 6 and 7, respectively. These chapters demonstrate the methodology on our first case study, a class representing chemical molecules. A second case study, a class representing mathematical graphs, is described in Chapter 8. Finally, our conclusions and directions for future research are discussed in Chapter 9. A literature survey of the applications of FCA to software engineering problem is provided in Appendix A.

# Chapter 2

## Definitions of Accesses Between Fields and Methods

The *set of fields* of a class and the *set of methods* of a class, along with the binary relation of *fields accessed by methods* between these sets, are at the heart of our methodology. Although these notions are simple and intuitive, we will come across some subtle intricacies when giving them precise definitions in the context of a concrete programming language. In this chapter, we discuss the definitions of these terms for the JAVA programming language.

This chapter is organized as follows: Sec. 2.1 defines the *set of fields* of a class. Sec. 2.2 defines the *set of methods*. In Sec. 2.3 we describe the relation of accesses to fields and discuss the problems that prevent its precise calculation. Instead, we present a simple process for approximating the relation of accesses to fields in Sec. 2.4. Applications of current static analysis techniques to the precise calculation of this relation are outside the scope of this work.

### 2.1 The set of fields of a class

Consider the JAVA class `Pnt`<sup>1</sup>, listed in Fig. 2.1, which represents an element of a two-dimensional picture.

Class `Pnt` defines three fields: `x`, `y`, and `color`. Each of these fields constitutes an important part of the state of `Pnt` instances. We also know that the class `Object`, which is the superclass of `Pnt`, does not define any fields. Therefore, the *set of fields* of `Pnt3D` is simply  $\{x, y, color\}$ .

Consider now the class `Pnt3D`, which represents a three-dimensional point. As shown in Fig. 2.2, `Pnt3D` inherits from `Pnt`. A new field named `z` represents the third coordinate, and is accompanied by an *inspector* (a “field-get” method) and a *mutator* (a “field-set” method). A new method named `setXYZ` is used to set all three coordinates at once. The `draw` method and the constructor are overridden.

Defining the set of fields of class `Pnt3D` is more complicated than defining the set of fields of class `Pnt`. The three fields defined in `Pnt` (`x`, `y` and `color`) have **private** access, and therefore cannot be accessed directly by the methods which class `Pnt3D` defines. In fact, the developer of class `Pnt3D` is not allowed to rely on the existence of these fields. Nevertheless, if we want to fully

---

<sup>1</sup>This class has been designed for demonstrational purposes.

```

public class Pnt {
    private int x,y;
    private int color;
    public Pnt() { setXY(0,0); setColor(0); }
    public int getX() { return x; }
    public void setX(int newX) { x = newX; }
    public int getY() { return y; }
    public void setY(int newY) { y = newY; }
    public void setXY(int newX, int newY) { setX(newX); setY(newY); }
    public int getColor() { return color; }
    public void setColor(int newColor) { color = newColor; }
    public void draw() { Application.pset2D(getX(),getY(),getColor()); }
}

```

Figure 2.1: Source code of class Pnt

```

public class Pnt3D extends Pnt {
    private int z;
    public Pnt3D() { setXYZ(0,0,0); setColor(0); }
    public int getZ() { return z; }
    public void setZ(int newZ) { z = newZ; }
    public void setXYZ(int newX, int newY, int newZ)
        { setXY( newX, newY ); setZ(newZ); }
    public void draw() { Application.pset3D(getX(),getY(),getZ(),getColor() ); }
}

```

Figure 2.2: Source code of class Pnt3D

understand the `Pnt3D` class, we must be able to understand the implementation of every method which could be invoked on its instances. Therefore, as some members of the interface of `Pnt3D` are defined in class `Pnt` and use its fields, we must include the fields of `Pnt` in the set of fields of class `Pnt3D`. The set of fields of `Pnt3D` is therefore:  $\{x, y, color, z\}$ . Even if `Pnt` had no methods, these fields would still be relevant during code inspection or reverse engineering.

Following is a formal definition of the set of fields of a class:

**Definition 1 (Set of fields)** *Let  $C$  be a class. The set of fields of  $C$  consists of all the fields which are defined in  $C$  or in any superclass of  $C$ . Fields of primitive types or of the `String` type which are declared as both **static** and **final** are excluded from this set.*

The definition excludes **final static** fields of a primitive type or of the `String` type, because they are nothing more than named constants within the class scope. It is also difficult to identify access operations to these fields in the class file representation, since the `JAVA` compiler relies on this property to inline their values. Note that **static non-final** fields are included in the definition, because they provide information that helps segregate methods, for reasons that are discussed in Chapter 4.

## 2.2 The set of methods of a class

The *set of methods* of `Pnt` consists of the eight methods defined in this class:

$\{\text{Pnt}, \text{getX}, \text{setX}, \text{getY}, \text{setY}, \text{setXY}, \text{getColor}, \text{setColor}, \text{draw}\}$ .

This set does *not* include the twelve methods inherited from class `Object`<sup>2</sup>, methods which deal with diverse issues such as synchronization, comparison, default printing, serialization, etc. These methods, which are present in every `JAVA` class, do not contribute to understanding the unique functionality of the class, and may clutter the class-specific information. It is also interesting that as many as six methods out of these are **final**, i.e., cannot be overridden at all. These methods can be thought of as general global utilities.

We are faced with a problem in classes which do not derive from `Object` directly: Should inherited methods be included in the set of methods of such a class? We argue that inherited methods must indeed be included for two reasons: First, these methods may constitute part of the external interface of the inheriting class (e.g. `setXY` which is inherited from `Pnt`). Second, the methods of the subclass may use the methods of the superclass (e.g. `setXYZ` which uses `setXY`).

In fact, we must include inherited methods even if they are overridden in the subclass. This is because `JAVA` allows an overriding method to invoke the overridden version using the **super** keyword. For example, because the constructor of `Pnt3D` initializes the `x` and `y` fields exactly like the constructor of `Pnt`, we can write it more economically as follows:

```
public Pnt3D() { super(); setZ(0); }
```

Later, after we calculate the *method calls* relation, we will be able to tell which overridden methods are actually used, and remove the rest.

---

<sup>2</sup>In version 1.4 of the `JDK`, the `Object` class defines the following methods:  $\{\text{clone}, \text{equals}, \text{getClass}, \text{finalize}, \text{hashCode}, \text{notify}, \text{notifyAll}, \text{registerNatives}, \text{toString}, \text{wait}(), \text{wait}(\text{long}), \text{wait}(\text{long}, \text{int})\}$ .

Note that when referring to specific methods and fields in this text, we borrow the `::` symbol from C++ to fully qualify names. The names `Pnt::draw` and `Pnt3D::draw` refer to the draw methods defined in the `Pnt` and `Pnt3D` classes, respectively.

We now give a precise definition of the set of methods of a class:

**Definition 2 (Set of methods)** *Let  $C$  be a class. The set of methods of  $C$  is the union of the sets of methods which are given a body or generated by the compiler in  $C$  and in every superclass of  $C$  except for `Object`.*

This definition encompasses compiler-generated methods such as generated default constructors. It does so because such methods can access fields and be invoked by other methods just like normal methods. In fact, although these methods do not exist in a JAVA source file, they appear in the binary classfile and their bytecode is executed by the Java Virtual Machine.

Note that although the set of methods contains both **static** and non-**static** methods, it does not contain methods declared as **abstract**. Such methods are declared and not defined; they are not given a body nor is one generated for them.

## 2.3 The relation of fields accessed by methods

In order to perform concept analysis on a class, we need to calculate the binary relation of accesses between its set of methods and its set of fields.

Consider Table 2.1 which depicts the relation of *direct accesses* between the set of fields of class `Pnt3D` and its set of methods. A check-mark at the intersection of a row and a column indicates that the method represented by that column makes at least one *direct read access* or *direct write access* to the field represented by that row. The `getX` method of class `Pnt3D` is an example of a method that makes a direct read access (a direct access that retrieves the value of a field), whereas the `setX` method is an example of a method that makes a direct write access (a direct access that changes the value of a field).

		methods													
		<code>Pnt3D::draw()</code>	<code>Pnt3D::setXYZ()</code>	<code>Pnt3D::setZ()</code>	<code>Pnt3D::getZ()</code>	<code>Pnt3D::Pnt3D()</code>	<code>Pnt::draw()</code>	<code>Pnt::setColor()</code>	<code>Pnt::getColor()</code>	<code>Pnt::setXY()</code>	<code>Pnt::setY()</code>	<code>Pnt::getY()</code>	<code>Pnt::setX()</code>	<code>Pnt::getX()</code>	<code>Pnt::Pnt()</code>
fields	<code>Pnt::x</code>		✓	✓											
	<code>Pnt::y</code>				✓	✓									
	<code>Pnt::color</code>						✓	✓							
	<code>Pnt3D::z</code>												✓	✓	

Table 2.1: *Direct field-accesses* relation of the `Pnt3D` class

We argue that the relation of direct accesses is not sufficient for a meaningful concept analysis of the class. For example, the relation we saw in Table 2.1, is very sparse and therefore misleading: Although the connection between the inspector-mutator pairs for each field is genuine, all the other methods mistakenly appear to not use any fields. We can expect to see this problem in many classes, because it is a common OOP practice that higher-level methods do not access fields directly.

To accurately portray the use of fields by methods, the relation of accesses to fields must also include methods that access fields *indirectly*. We say that a method accesses a field indirectly if it starts a chain of method invocations in which one of the methods accesses the field directly. This approach is similar to that taken by Bieman and Kang [5] in calculating a metric for *class cohesion*. The complete field accesses relation for `Pnt3D`, which appears in Table 2.2, includes entries for direct and indirect accesses.

		methods													
		<code>Pnt3D::draw()</code>	<code>Pnt3D::setXYZ()</code>	<code>Pnt3D::setZ()</code>	<code>Pnt3D::getZ()</code>	<code>Pnt3D::Pnt3D()</code>	<code>Pnt::draw()</code>	<code>Pnt::setColor()</code>	<code>Pnt::getColor()</code>	<code>Pnt::setXY()</code>	<code>Pnt::setY()</code>	<code>Pnt::getY()</code>	<code>Pnt::setX()</code>	<code>Pnt::getX()</code>	<code>Pnt::Pnt()</code>
fields	<code>Pnt::x</code>	✓	✓	✓			✓			✓	✓			✓	✓
	<code>Pnt::y</code>	✓			✓	✓	✓			✓	✓			✓	✓
	<code>Pnt::color</code>	✓					✓	✓	✓	✓	✓			✓	✓
	<code>Pnt3D::z</code>											✓	✓	✓	✓

Table 2.2: *Field-accesses* relation of the `Pnt3D` class

Because this work deals with proposing an application of the field accesses relation, the details of its exact calculation are less important. The problem of discovering accesses to data entries, including indirect accesses to fields, is an active research area in static analysis and depends on the intricacies of the programming language. As the quality of static analysis techniques improve, we expect that the precision and hence the applicability of our methodology will increase.

In the interest of clarity, the following section presents a very simple technique for approximating the field accesses relation of a JAVA class. In the remainder of this work, we assume that all lattices are constructed upon an access relation that was calculated by a precise static analysis technique.

## 2.4 Approximating the field accesses relation

In the interest of completeness, the rest of this chapter presents and demonstrates a simple technique for calculating an approximation of the field accesses relation of a JAVA class. We first present the simplifications upon which this technique is based, and then proceed to describe the calculation process in detail.

## 2.4.1 Simplifications

Our technique for approximating the field accesses relation is based on the following five simplifications:

**Simplification 1 (Ignoring other classes)** *Only the files for the class under analysis and its super-classes are processed. All other classes are ignored.*

This simplification has three major advantages:

1. The number of files which must be processed is limited to a few classes.
2. The relation of direct calls over which we must calculate a transitive closure is limited to the set of methods of the class under analysis. Discovering chains of method calls involves the calculation of a transitive closure over a relation of direct calls to methods. Since this is an expensive operation, limiting the size of the relation significantly reduces computation time.
3. The targets of fewer virtual calls have to be resolved.

Unfortunately, this simplification also has an adverse effect on the precision of the resulting relation. Consider for example, the following JAVA classes:

```
public class A {
    private static int field1;
    public static void method1() { B.method3(); }
    public static void method2() { field1=0; }
}

public class B {
    public static void method3() { A.method2(); }
}
```

If class B is not processed, the chain of calls from `method1` to `method2` via `method3` is not be discovered, and therefore the field accesses relation would not indicate that `method1` accesses the `field1` field.

**Simplification 2 (Using classfiles)** *JAVA classfiles are used instead of JAVA source files.*

JAVA classfiles are easier to analyze than regular JAVA source files. They also allow the use of our methodology for reverse-engineering classes in the absence of source files. The disadvantage of this simplification is that the relation omits some details, such as the use of constants in the form of `final static` fields of primitive types.

**Simplification 3 (Ignoring field aliases)** *Accesses to fields using aliases are ignored.*

Suppose that at some point in the execution of the program a variable,  $v$ , and a member field,  $f$ , both refer to some object,  $o$ . In our approximation, only methods that explicitly use  $f$  are considered as accessing that field; methods that only use  $v$  do not. This approach greatly simplifies our analysis, at the cost of missing some accesses. For example, our approximation reports for the following code that `method1` accesses field `field1` but that `method2` does not:

```

public class C {
    private Object field1;
    public void method1() { method2(field1); }
    public void method2(Object obj) { System.out.println(obj); }
}

```

Note that whereas C++ allows the creation of aliases to fields via the use of references, no equivalent construct exists in JAVA. In other words, it is not possible to make a certain field refer to a different object unless the name of that field is explicitly used.

**Simplification 4 (Resolving targets of virtual calls)** *Targets of virtual calls to methods on an object whose static type is the class under analysis or one of its superclasses, are resolved as if the dynamic type of that object is the class under analysis.*

Because of the dynamic dispatching mechanism, it is impossible to deduce in advance which version of a method will be invoked in response to a certain call. This fact complicates the calculation of the accesses relation, which uses chains of method calls to discover indirect accesses to fields. For example, consider Fig. 2.3 which depicts a class hierarchy consisting of a base class and three deriving classes.

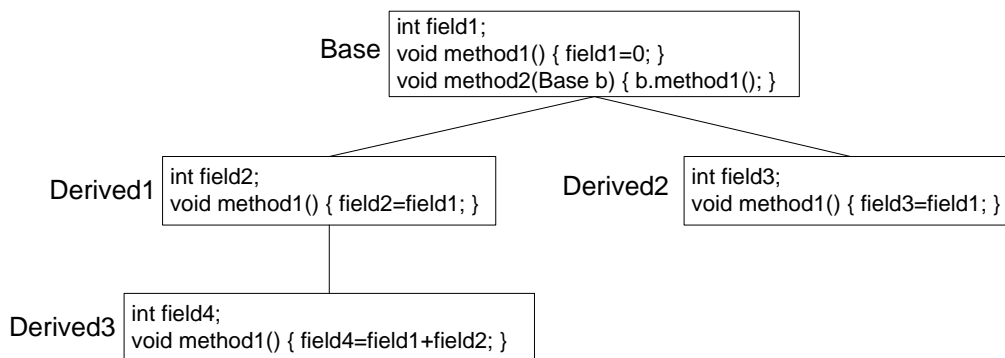


Figure 2.3: A class hierarchy where dynamic dispatching affects the accesses relation

If the class under analysis is `Base1`, then it is straightforward to infer that `method1` uses both `field1` and `field2`. However, in order to determine the fields used by `method2`, we need to decide which version of `method1` is actually invoked. This depends, of course, on the dynamic type of the parameter `b`, which can be each of the four classes.

One way to resolve this problem is to arbitrarily consider the method as invoking all the versions in all the potential targets, but we reject this option for several reasons. First, selecting all the targets might lead to an explosion in the number of invocation chains. Second, some of the targets may be foreign to the class under analysis and have different fields (e.g., `Derived2`). Third, some potential targets can be in subclasses of the class under analysis (e.g., `Derived3`), and will therefore be ignored due to Simplification 1.

Instead, we choose to arbitrarily select a single target. Since the analysis of a particular class is aimed at understanding how an instance of that class operates, we decide that every object which can be an instance of the class under analysis is treated as such. Hence, in analyzing `derived1`, we assume that the type of `b` is also `derived1` and therefore `method2` uses both `field1` and `field2`.

**Simplification 5 (Native methods are not analyzed)** *Because of the limitations of simple JAVA static analyzers, native methods are not analyzed, and are considered as accessing no fields.*

The five simplifications made by our technique allow it to be described and calculated easily, as described in the following subsection.

## 2.4.2 Calculation process

The calculation process of the approximation begins with a specialized JAVA-classfile disassembler which reads the class files of the class under analysis and all its superclasses. The disassembler collects the set of methods and the set of fields of the class. It then creates for each method a list of fields that the method accesses directly and a list of methods that it calls directly. These lists form the relations of *direct accesses to fields* and *direct calls to methods*, which are used to calculate the complete relation of accesses to fields.

The relation of direct accesses to fields is based on the following definition of a direct access to a field:

**Definition 3 (Direct access to a field)** *A direct access to a field appears in a JAVA source file in one of the following two forms:*

**Direct read** *The field serves as an rvalue: The name of the field appears in an expression which is not an assignment, or on the right hand side of an assignment expression. During the evaluation of the expression, the value of the field is retrieved.*

**Direct write** *The field serves as an lvalue: The name of the field appears on the left hand side of an assignment expression. During the evaluation of the expression, the value of the field is (potentially) modified.*

*In the bytecode of an accessing method in a compiled JAVA class, a direct read access appears as a `getfield` or `getstatic` opcode, and a direct write appears as a `putfield` or `putstatic` opcode. These opcodes operate on an object whose dynamic type can be the class under analysis.*

The relation of direct accesses to fields for the `Pnt3D` class appears in Table 2.1. In order to calculate indirect accesses to fields, information about calls to methods must be used. Table 2.3 depicts the relation of direct method calls of class `Pnt3D`. Because of simplification 1, this relation involves only the set of methods of class `Pnt3D`.

In Table 2.3, a check-mark at the intersection of a row and a column indicates that there is at least one *direct call* from the method represented by that column to the method represented by that row.

**Definition 4 (Direct call to a method)** *In the bytecode of an invoking method in a compiled JAVA class, a direct call to a method appears as one of the method invocation opcodes: `invokevirtual`, `invokestatic`, and `invokespecial`. These opcodes operate on an object whose dynamic type can be the class under analysis, and targets of virtual calls are resolved accordingly.*

Because Simplification 1 allows us to ignore chains of calls that include methods from other classes, we can obtain the complete relation of method calls by calculating the transitive closure of

		calling methods												
		Pnt3D::draw()	Pnt3D::setXYZ()	Pnt3D::setZ()	Pnt3D::getZ()	Pnt3D::Pnt3D()	Pnt::draw()	Pnt::setColor()	Pnt::getColor()	Pnt::setXY()	Pnt::setY()	Pnt::getY()	Pnt::setX()	
called methods	Pnt::Pnt()													
	Pnt::getX()						✓						✓	
	Pnt::setX()								✓					
	Pnt::getY()										✓			
	Pnt::setY()								✓					
	Pnt::setXY()	✓											✓	
	Pnt::getColor()							✓						✓
	Pnt::setColor()	✓												
	Pnt::draw()													
	Pnt3D::Pnt3D()													
	Pnt3D::getZ()													✓
	Pnt3D::setZ()												✓	
	Pnt3D::setXYZ()													✓
	Pnt3D::draw()													

Table 2.3: *Direct method calls* relation of the Pnt 3D class

		calling methods												
		Pnt3D::draw()	Pnt3D::setXYZ()	Pnt3D::setZ()	Pnt3D::getZ()	Pnt3D::Pnt3D()	Pnt::draw()	Pnt::setColor()	Pnt::getColor()	Pnt::setXY()	Pnt::setY()	Pnt::getY()	Pnt::setX()	Pnt::Pnt()
called methods	Pnt::Pnt()													
	Pnt::getX()						✓							✓
	Pnt::setX()	✓							✓				✓	
	Pnt::getY()									✓				✓
	Pnt::setY()	✓							✓					
	Pnt::setXY()	✓												✓
	Pnt::getColor()										✓			✓
	Pnt::setColor()	✓												
	Pnt::draw()													
	Pnt3D::Pnt3D()													
	Pnt3D::getZ()													✓
	Pnt3D::setZ()													✓
	Pnt3D::setXYZ()													✓
	Pnt3D::draw()													

Table 2.4: *Method-calls* relation of the Pnt3D class

the relation of direct method calls. The complete relation of method calls of class `Pnt3D` appears in Table 2.4.

Using the relation of method calls of `Pnt3D` from Table 2.4 and the relation of direct accesses to fields from Table 2.1, we can calculate the complete relation of field accesses which appeared in Table 2.2. In the complete relation, there is an entry for a method  $m$  and a field  $f$ , if there is such an entry in the direct relation, or if there is a method  $m'$  such that  $m'$  accesses  $f$  directly and there is a chain of calls from  $m$  to  $m'$ .

# Chapter 3

## Review of Formal Concept Analysis

The sets of fields and methods of a class and the binary relation of accesses between them constitute a particular *formal context*. In order to extract information from a context, we perform *formal concept analysis* and obtain a set of *concepts*. These concepts are presented in a graph called a *concept lattice*.

This chapter, in which all the above notions are given a precise definition, serves as a primer on the basics of concept analysis. In addition, we demonstrate how concept analysis is performed on the running example of class `Pnt3D`, which was introduced in the previous chapter. The interpretation of the resulting concept lattice is left for the next chapter.

The organization of this chapter is as follows: Sec. 3.1 defines *formal contexts* and presents the *complete context*, a specific formal context which is based on the field accesses relation and used throughout this work. Sec. 3.2 defines *formal concepts*. *Concept lattices* are defined in Sec. 3.3.

### 3.1 Formal contexts

Consider again the field accesses relation of the `Pnt3D` class, which appeared in Table 2.2. For every member of the set of fields of class `Pnt3D`, this relation specifies the members of the set of methods which access that field. We can think of every member of the set of fields as an *object* (not to be confused with the OOP term). The access that a certain method makes to a certain field can then be thought of as an *attribute* of the corresponding object: The field has the property that it is accessed by the specific method.

Together, the set of methods of the class, the set of fields of the class, and the accesses relation between them constitute a specific *formal context* which we call the *complete context*. A formal context (or simply a “context”) connects a set of *objects* and a set of *attributes* by specifying the attributes that each object has. In the complete context of a class, the set of objects is the set of fields, the set of attributes is the set of methods, and the relation specifies for each field the methods which access it.

Table 2.2, which depicts the relation of accesses to the fields of class `Pnt3D`, is also the table of the complete context of that class, as can be seen in Table 3.1.

In Table 3.1, a row corresponds to each object (field), and the entries in that row correspond to the attributes (methods) owned by that object. We say that a method *uses a combination of fields* if it uses all the fields in that combination and no others. The table column for that method would then

		attributes												
		Pnt3D::draw()	Pnt3D::setXYZ()	Pnt3D::setZ()	Pnt3D::getZ()	Pnt3D::Pnt3D()	Pnt::draw()	Pnt::setColor()	Pnt::getColor()	Pnt::setXY()	Pnt::setY()	Pnt::getY()	Pnt::setX()	Pnt::getX()
objects	Pnt::x	✓	✓	✓			✓			✓	✓		✓	✓
	Pnt::y	✓			✓	✓	✓			✓	✓		✓	✓
	Pnt::color	✓						✓	✓	✓	✓			✓
	Pnt3D::z											✓	✓	✓

Table 3.1: Complete context of the Pnt 3D class

have entries exactly for the fields in the combination.

Following are precise definitions of a *context* and of the *complete context*.

**Definition 5 (Context)** A context (or “formal context”) is denoted by a triple  $\langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$ . In this triple,  $\mathbf{O}$  is a set of objects (also known in the literature as instances or extent),  $\mathbf{A}$  is a set of attributes (also known as features or intent), and  $\mathbf{R}$  is a binary relation between them. For any  $\langle o, a \rangle \in \mathbf{R}$  we say that object  $o$  has or owns attribute  $a$ , and that attribute  $a$  belongs to or is owned by  $o$ .

**Definition 6 (Complete context of a class)** Let  $C$  be a class, and let  $\mathbf{O}_C$  and  $\mathbf{A}_C$  respectively be its set of fields and its set of methods. In the complete context, the set of objects is  $\mathbf{O}_C$ , the set of attributes is  $\mathbf{A}_C$  and a pair  $\langle o \in \mathbf{O}_C, a \in \mathbf{A}_C \rangle$  is in the relation of the context if and only if the method denoted by  $a$  accesses the field denoted by  $o$ .

The complete context is not the only context which can be obtained from the relation of accesses to fields. For example, we can create a context without the overridden methods (e.g. `Pnt::Pnt` and `Pnt::draw` in class `Pnt3D`). This particular context, which we call the *concrete context* is identical to the complete context except that the columns for these methods are removed. We discuss the relations between contexts and present additional criteria for creating contexts in Chapter 4.

## 3.2 Formal concepts

### 3.2.1 Galois Connection

A context can be used to discover the *common attributes* of a subset of its objects, and the *common objects* of a subset of its attributes.

Let  $\langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$  be a context. For a given nonempty subset of objects,  $O \subseteq \mathbf{O}$ , the set of *common attributes*, denoted  $\overline{O}$ , consists of every attribute in  $\mathbf{A}$  which is in the relation with every object in  $O$ . More formally,  $\overline{O} = \{a \in \mathbf{A} \mid \forall o \in O : (o, a) \in \mathbf{R}\}$ . For example, the set of common attributes of the set  $\{x, y, color\}$  in Table 3.1 is  $\{Pnt3D, draw\}$ .

In a similar manner, for a given nonempty subset of attributes,  $A \subseteq \mathbf{A}$ , the set of *common objects*, denoted  $\overline{A}$ , consists of every object in  $\mathbf{O}$  which is in the relation with every attribute in  $A$ :  $\overline{A} = \{o \in \mathbf{O} \mid \forall a \in A : (o, a) \in \mathbf{R}\}$ . Hence, the set of common objects of the set of attributes  $\{\text{Pnt3D}, \text{draw}\}$  is  $\{x, y, z, \text{color}\}$ .

For empty sets, the definition is extended: The set of common attributes of the empty set of objects is  $\mathbf{A}$ , and the set of common objects of the empty set of attributes is  $\mathbf{O}$ .

The notions of common attributes and common objects give rise to a special connection between the sets of objects and attributes of a class: The definition of common attributes forms a mapping from the power set of  $\mathbf{O}$  to the power set of  $\mathbf{A}$ , and the definition of common objects forms a mapping from the power set of  $\mathbf{A}$  to the power set of  $\mathbf{O}$ . Together, these two mappings form a *Galois connection* between the objects and attributes of the context.

### 3.2.2 Concepts

For the *concrete context* of class `Pnt3D`, presented in Table 3.2, consider the subset of objects  $O = \{x, y\}$  and the subset of attributes  $A = \{\text{setXY}, \text{setXYZ}, \text{draw}, \text{Pnt3D}\}$ .

		attributes									
		Pnt3D::draw()	Pnt3D::setXYZ()	Pnt3D::setZ()	Pnt3D::getZ()	Pnt3D::Pnt3D()	Pnt::setColor()	Pnt::setColor()	Pnt::getColor()	Pnt::setXY()	Pnt::setX()
objects	Pnt::x	✓	✓			✓			✓		✓
	Pnt::y			✓	✓	✓			✓		✓
	Pnt::color						✓	✓	✓		✓
	Pnt3D::z								✓	✓	✓

Table 3.2: *Concrete context* of the `Pnt3D` class

If we calculate the sets of common attributes and common objects for these two sets, we notice an interesting phenomenon: The set of common attributes of  $O$  is exactly  $A$ , and the set of common attributes of  $A$  is exactly  $O$ . This phenomenon is captured by the notion of a *formal concept*, a pair consisting of a set of objects and a set of attributes which are mapped into each other by the Galois connection.

**Definition 7 (Formal concept)** *Given a context  $\langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$  and two subsets,  $O \subseteq \mathbf{O}$  and  $A \subseteq \mathbf{A}$ , we say that the pair  $\langle O, A \rangle$  is a formal concept (or simply a concept) if  $\overline{O} = A$  and  $\overline{A} = O$  (or simply:  $\overline{\overline{O}} = O$ ).*

In other words, if the pair is a concept, then every object of  $O$  has every attribute in  $A$ , and every attribute in  $A$  is owned by every object in  $O$ . Note that the sets  $O$  and  $A$  are *maximal*: there is no other

attribute  $a \in \{\mathbf{A} \setminus A\}$  which is owned by every object in  $O$ , and there is no other object  $o \in \{\mathbf{O} \setminus O\}$  which owns every attribute in  $A$ .

Table 3.3 lists all the concepts for the concrete context of `Pnt3D`. The algorithm for obtaining concepts from a context is discussed at the end of this chapter.

Name (arbitrary)	Objects	Attributes
Concept #1	{ }	{ getX, setX, getY, setY, setXY, getColor, setColor, Pnt3D, getZ, setZ, setXYZ, draw }
Concept #2	{ color }	{ getColor, setColor, Pnt3D, draw }
Concept #3	{ x }	{ getX, setX, setXY, Pnt3D, setXYZ, draw }
Concept #4	{ y }	{ getY, setY, setXY, Pnt3D, setXYZ, draw }
Concept #5	{ x, y }	{ setXY, Pnt3D, setXYZ, draw }
Concept #6	{ z }	{ Pnt3D, getZ, setZ, setXYZ, draw }
Concept #7	{ x, y, z }	{ Pnt3D, setXYZ, draw }
Concept #8	{ x, y, color, z }	{ Pnt3D, draw }

Table 3.3: Concepts for the concrete context of the `Pnt3D` class

If a certain combination of fields is used by at least one method, then a corresponding concept must exist with the appropriate set of objects (but a more elaborate set of attributes). For example, the combination of fields  $\{x,y\}$ , which is the exact combination used by the `setXY` method, is also the set of objects of concept  $C_5$ . Note that the converse is not true: It is possible for a concept to exist without a corresponding combination of fields being used<sup>1</sup>. Nevertheless, in general we will not see concepts for unused combinations. For example, consider the following combination of fields:  $\{x,y,color\}$ . The set of methods which use all the fields of this combination is exactly its set of common attributes:  $\{Pnt3D, draw\}$ . The exact combination of fields used by these two methods is the set of their common objects:  $\{x,y,z,color\}$ . Hence, the Galois connection does not map each set into the other, and there is no concept for this combination.

The maximal number of concepts for a certain context is bound by the number of objects and attributes in that context. Each concept must have a unique subset of objects and a unique subset of attributes out of the sets of objects and attributes of the context. If a context has  $|\mathbf{O}|$  objects and  $|\mathbf{A}|$  attributes, then there are  $2^{|\mathbf{O}|}$  combinations of objects and  $2^{|\mathbf{A}|}$  combinations of attributes. Hence, there can be at most  $2^{\min(|\mathbf{O}|, |\mathbf{A}|)}$  concepts.

We can expect organized classes to have a relatively small number of concepts, because the methods of an organized class tend to use a limited number of field combinations. For example, there is evidence of some organization in `Pnt3D`: Although the power set of its objects consists of 16 combinations, only 8 combinations have corresponding concepts. Interestingly, the number of concepts in the lattices of actual classes is often linear in  $\min(|\mathbf{O}|, |\mathbf{A}|)$ .

Consider the scatter-graph in Fig. 3.1 which plots the number of concepts against the minimum of  $\mathbf{O}$  and  $\mathbf{A}$  for more than 3,000 classes taken from the JDK<sup>2</sup>. The dashed lines indicate multiplicands (from 1 to 7) of  $\min(|\mathbf{O}|, |\mathbf{A}|)$ . A similar graph for more than 3000 other classes taken from the

<sup>1</sup>In the sparse representation, presented later, such concepts introduce no attributes.

<sup>2</sup>We took classes from all the `.jar` files in version 1.4.1 of the JDK.

eclipse<sup>3</sup> framework appears in Fig. 3.2. In the majority of classes in both data sets, the number of concepts falls below three times the minimum, and the rest fall below seven times the minimum. Hence, the number of concepts in these data sets is linear and not exponential in the minimum of fields and methods.

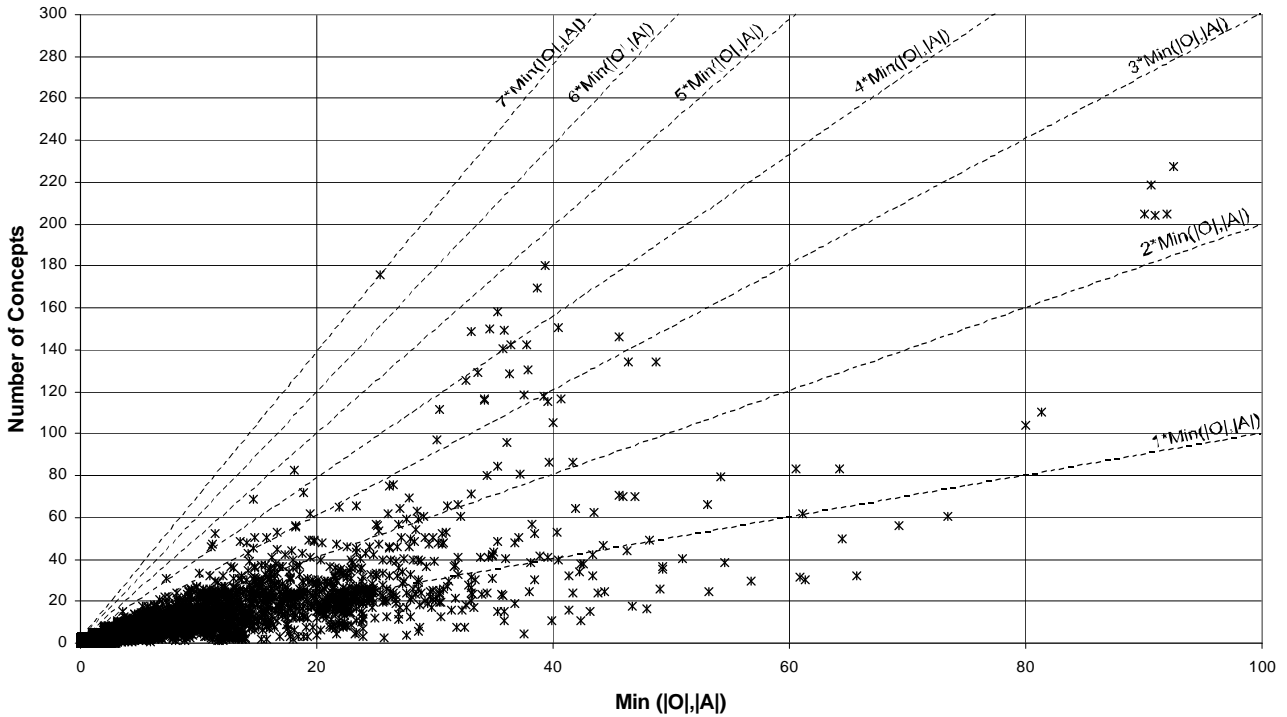


Figure 3.1: A scatter graph showing the number of concepts relative to  $\min(|\mathbf{O}|, |\mathbf{A}|)$  in the JDK ( $\mathbf{O}$  and  $\mathbf{A}$  respectively denote the sets of fields and methods of a class)

Of course, for our partitioning of the methods into equivalence classes to be useful, each concept should contain more than one method. Fig. 3.3 and Fig. 3.4 show that this is indeed the case for most classes: the number of concepts for a class is usually lower than the number of methods in that class. The average number of methods per concept will increase further if we limit ourselves only to concepts that introduce methods (contrary to counting empty concepts and concepts that introduce only fields, as is done in the figures).

### 3.2.3 Order between concepts

In Table 3.3, which lists the concepts of class `Print3D`, we see an inverse relation between the number of objects and the number of attributes of each concept. If one concept contains a superset of the objects of another concept, then that first concept also contains a subset of the attributes of the second. Similarly, if a concept contains a superset of the attributes of another, it also contains a subset of the objects of the other. This inverse relation allows the definition of a partial order between concepts:

**Definition 8 (Domination between concepts)** *Let  $c_1 = \langle O_1, A_1 \rangle$  and  $c_2 = \langle O_2, A_2 \rangle$  be two concepts. If  $O_1 \supset O_2$  and  $A_1 \subset A_2$ , then concept  $c_1$  dominates concept  $c_2$ ; this domination is de-*

<sup>3</sup>We took classes from all the `.jar` files (including third-party archives) in release 2.0.2 of the *eclipse framework* [15].

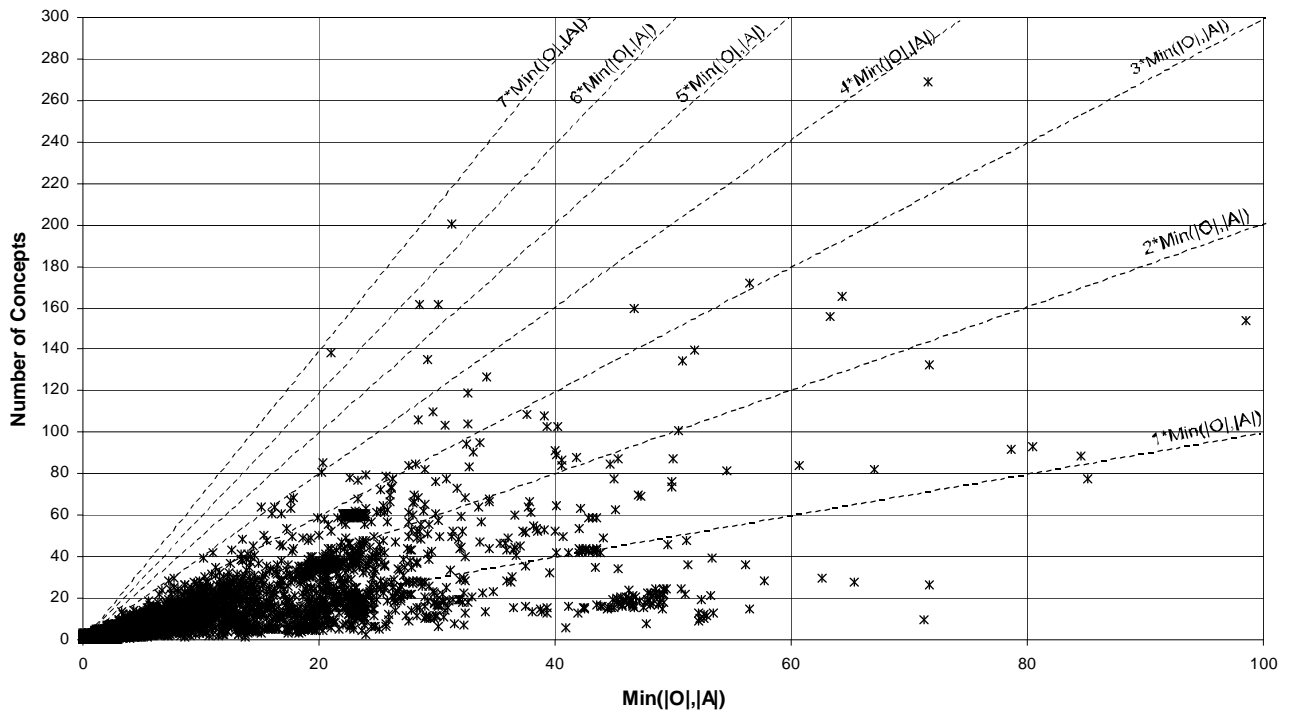


Figure 3.2: A scatter graph showing the number of concepts relative to  $\min(|\mathbf{O}|, |\mathbf{A}|)$  in *Eclipse* ( $\mathbf{O}$  and  $\mathbf{A}$  respectively denote the sets of fields and methods of a class)

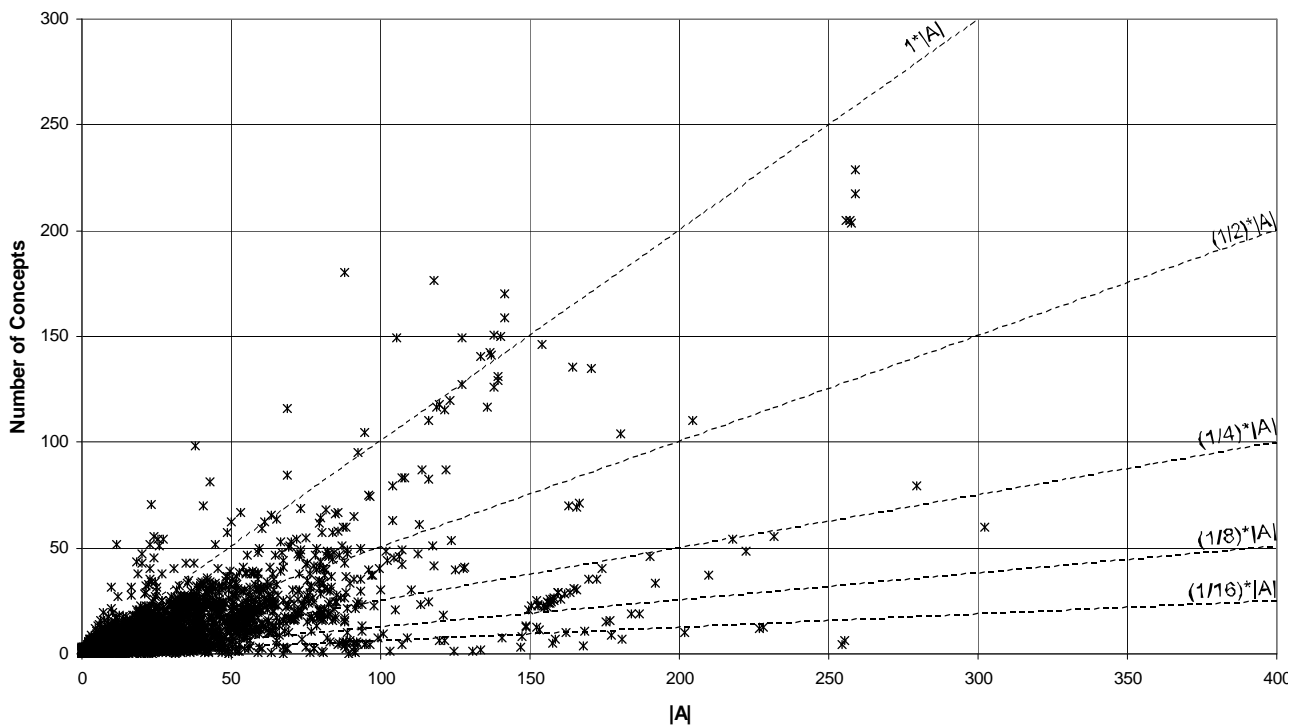


Figure 3.3: A scatter graph showing the number of concepts relative to  $|\mathbf{A}|$  in the JDK ( $\mathbf{A}$  denotes the set of methods of a class)

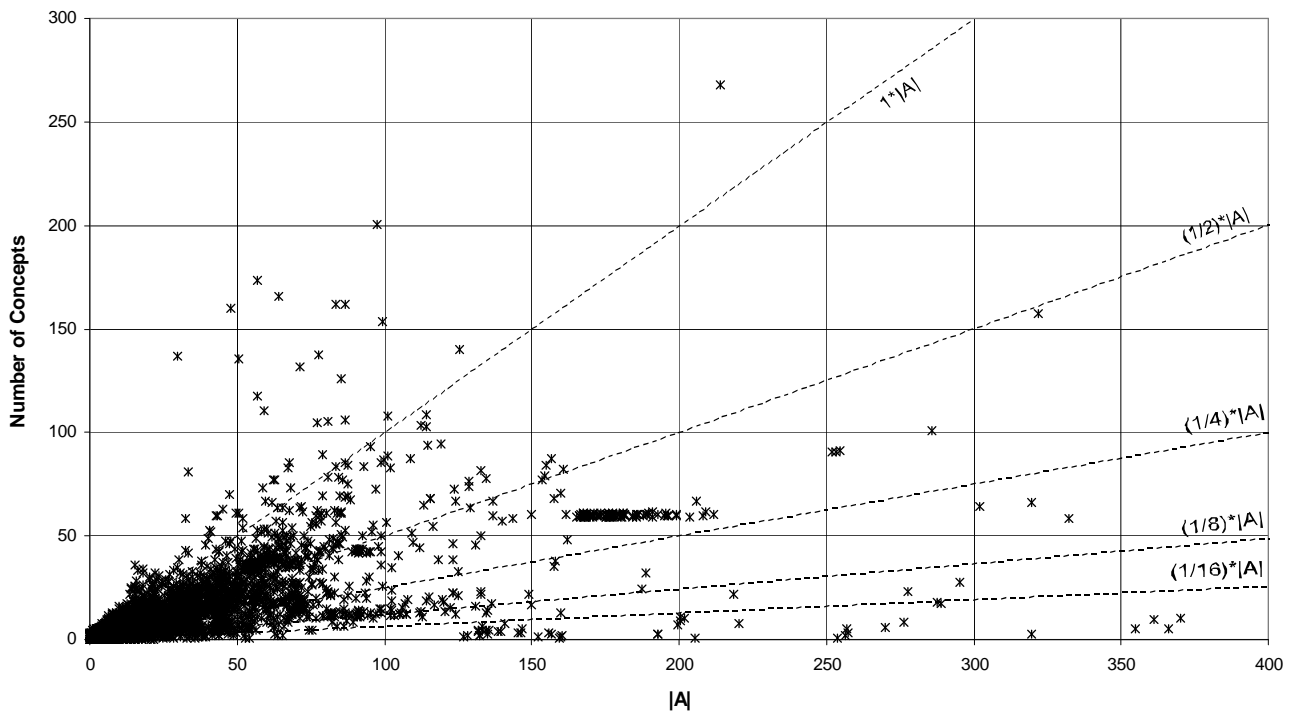


Figure 3.4: A scatter graph showing the number of concepts relative to  $|A|$  in *Eclipse* ( $A$  denotes the set of methods of a class)

noted  $c_1 > c_2$ . If there is no third concept,  $c_3$ , such that  $c_1 > c_2$  and  $c_2 > c_3$ , then  $c_1$  dominates  $c_2$  directly.

For example, concept  $C_7$  in Table 3.3, whose objects are  $\{x,y,z\}$ , dominates concept  $C_5$ , whose objects are  $\{x,y\}$ . Concept  $C_5$  dominates concepts  $C_3$  and  $C_4$  whose sets of objects are  $\{x\}$  and  $\{y\}$  respectively. It follows that concept  $C_7$  also dominates concepts  $C_3$  and  $C_4$ , but does not dominate them directly.

The order implied by the notion of domination between concepts is (except for trivial cases) a partial order. For example, it is impossible to compare between concept  $C_3$ , whose only object is  $x$ , and concept  $C_4$ , whose only object is  $y$ . Because partial orders are harder to understand than full orders, visualization using a *Hassé diagram* diagram can be of use. The partial order between concepts is visualized using a special *Hassé diagram* called a *concept lattice*.

### 3.3 Concept lattices

Fig. 3.5 depicts the concept lattice of the concrete context of class `Pnt 3D`.

The concept lattice in Fig. 3.5 has a vertex for every concept from Table 3.3. Each vertex is fully annotated: It lists all the objects and attributes of that concept. An edge connects two concepts if one dominates the other directly. We choose a vertical layout for the lattice so that each concept appears higher than any other concept that it dominates.

The lattice in Fig. 3.5 has an interesting property: For every two concepts we choose, either one

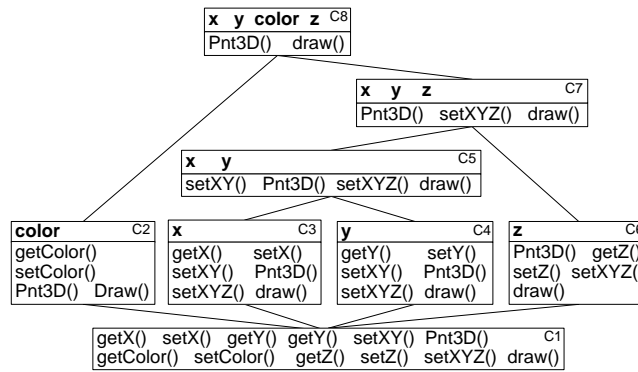


Figure 3.5: Fully annotated concept lattice of the concrete context of class Pnt3D

dominates the other, or there exists a third concept which dominates both concepts and also a fourth concept which is dominated by both concepts. This property is in fact a property of all concept lattices, and is stated in Wille’s *fundamental theorem on concept lattices* [22, 60]. According to this theorem, every concept lattice is a *complete lattice*. In other words, every two concepts in a concept lattice have a common *infimum* (greatest common subconcept) and a common *supremum* (greatest common superconcept). The objects of the infimum of two concepts are the intersection of the objects of both concepts; its attributes are the common attributes of its objects. Similarly, the attributes of the supremum of two concepts are the intersection of the attributes of both concepts; its objects are the common objects of its attributes.

In Fig. 3.5, for example, the infimum of concepts  $C_3$  and  $C_4$  is concept  $C_1$ , and their supremum is concept  $C_5$ . The infimum of concepts  $C_2$  and  $C_5$  is concept  $C_1$  and their supremum is concept  $C_8$ .

Note that Wille’s theorem implies that the supremum and the infimum of two concepts can be one of the two. Let  $c_1$  be a concept which dominates another concept  $c_2$ , so that the objects of  $c_2$  are a strict subset of the objects of  $c_1$ . The intersection of the objects of  $c_1$  and  $c_2$ , which constitutes the set of objects of their infimum, is therefore  $c_2$ . Because the set of objects of each concept must be unique, it follows that the infimum of  $c_1$  and  $c_2$  is  $c_2$ . Similarly, the supremum of both concepts is  $c_1$ .

Because every two concepts have a supremum and an infimum, the lattice itself has a single infimum and a single supremum. The supremum of the lattice, which we call the *top concept*, dominates every other concept in the lattice. It contains more objects and less attributes than any other concept. The top concept of the lattice in Fig. 3.5 is concept  $C_8$ . Similarly, the infimum of the lattice, which we call the *bottom concept*, is dominated by every other concept and therefore contains less objects and more attributes than any other concept. The bottom concept of the lattice in the figure is concept  $C_1$ .

### 3.3.1 Sparse annotation

Consider again the lattice in Fig. 3.5. Along any ascending path from the bottom concept to the top concept, each concept contains more objects than its predecessor. For example, the set of objects of concept  $C_5$  is the union of the sets of objects of the concepts it dominates directly, concepts  $C_3$  and  $C_4$ . Similarly, the set of objects of concept  $C_7$  is the union of the objects of concepts  $C_5$  and  $C_6$ , and the set of objects of concept  $C_8$  is the union of the objects of concepts  $C_2$  and  $C_7$ . We say that concepts such as  $C_5$ ,  $C_7$  and  $C_8$  *combine* the objects of the concepts they dominate directly.

Not all concepts combine the objects of the concepts they dominate. Concepts  $C_2$ – $C_4$  and concept  $C_6$  have at least one object, whereas concept  $C_1$ , which all of them dominate, has none. We say that such concepts *introduce* new objects.

In a similar manner, on any descending path from the top concept to the bottom concept, each concept contains more attributes than its predecessors. Concepts can combine the attributes of concepts which directly dominate them, or introduce new ones.

Here is a precise definition of the notion of introducing objects and attributes:

**Definition 9 (Introduced objects and attributes)** *Let  $C$  be a concept. An object which appears in  $C$  but not in any concept dominated by  $C$  is introduced in  $C$ . An attribute which appears in  $C$  but not in any concept dominating  $C$  is introduced in  $C$ .*

The main implication of Definition 9 is that every object and attribute is introduced in exactly one concept. This implication is at the heart of the convention of *sparse annotation* (or *inheritance convention* [26]) of concept lattices: Every concept is annotated only with the objects and attributes it introduces. Fig. 3.6 depicts the concept lattice from Fig. 3.5 using this convention.

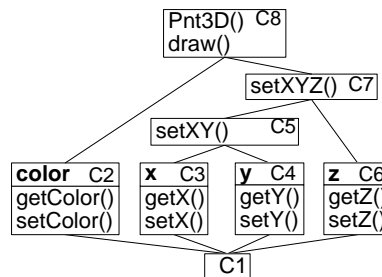


Figure 3.6: Concept lattice of the concrete context of class `Pnt3D`, using sparse annotations.

In the remainder of this thesis, all lattices are depicted using the sparse annotation. This decision is valid because a lattice with sparse annotations is equivalent to a lattice with full annotations. There is a simple process for translating between the two conventions.

To create the sparse annotation of a lattice from its full annotation, we go over each concept and calculate the objects and attributes it introduces. If a concept,  $c$ , introduces an object, then that object does not appear in the full annotation of any concept which is dominated directly by  $c$ . Similarly, if  $c$  introduces an attribute, then that attribute does not appear in the full annotation of any concept which dominates  $c$  directly.

It is also possible to compute the full annotation from the sparse annotation. The set of objects in the full annotation of  $c$  is the union of the sets of objects introduced in  $c$  and in every concept that  $c$  dominates. The set of attributes in the full annotation of  $c$  is the union of the sets of attributes introduced in  $c$  and in every concept that dominates  $c$ .

### 3.3.2 An algorithm for computing concepts and constructing lattices

Various algorithms and methods exist for the computation of formal concepts and the construction of their lattices [4, 36, 43]. We describe here a simple bottom-up worklist-based algorithm and demon-

strate it on the small Pnt class from Chapter 2, whose complete context appears in Table 3.4. The same algorithm is discussed and demonstrated in more detail by Siff and Reps [48].

		attributes								
		Pnt()	getX()	setX()	getY()	setY()	setXY()	getColor()	setColor()	draw()
objects	x	✓	✓	✓			✓			✓
	y	✓			✓	✓	✓			✓
	color	✓						✓	✓	✓

Table 3.4: Complete context of the Pnt class

The construction algorithm consists of three base steps and a fourth closure step. In all four steps, every newly-discovered concept is given a serial number according to the time of its discovery.

**Step 1 (Calculate the bottom concept of the lattice)** *The set of attributes of the bottom concept is the set of common attributes of the empty set of objects. The set of common objects of these attributes becomes the set of objects of the bottom concept.*

For the Pnt class, the set of common attributes of the empty set is the set of all nine methods, and the common objects of these attributes is the empty set. The bottom concept is therefore:

$$C_0 = \langle \{ \}, \{ \text{Pnt}, \text{getX}, \text{setX}, \text{getY}, \text{setY}, \text{setXY}, \text{getColor}, \text{setColor}, \text{draw} \} \rangle$$

**Step 2 (Calculate the atomic concepts)** *The atomic concepts are concepts which directly dominate the bottom concept. An atomic concept is created by taking a set that consists of a single object, calculating its set of common attributes, and then calculating the set of common objects of these attributes.*

The set of common attributes of the set {x} is {Pnt,getX,setX,setXY,draw}. The set of common objects of these attributes is again the set {x}. The atomic concept for x is therefore:

$$C_1 = \langle \{x\}, \{ \text{Pnt}, \text{getX}, \text{setX}, \text{setXY}, \text{draw} \} \rangle$$

The atomic concepts for y and color are calculated in the same manner:

$$C_2 = \langle \{y\}, \{ \text{Pnt}, \text{getY}, \text{setY}, \text{setXY}, \text{draw} \} \rangle$$

$$C_3 = \langle \{color\}, \{ \text{Pnt}, \text{getColor}, \text{setColor}, \text{draw} \} \rangle$$

Note that the number of atomic concepts may be smaller than the number of objects if several objects have the same set of common attributes.

**Step 3 (Initialize a list unordered pairs of concepts)** *We create a list of unordered pairs of concepts, and initialize it with all the foreign atomic concepts (atomic concepts that do not dominate each other).*

Because none of the three atomic concepts dominate each other, all the combinations appear in the list:  $L = \langle (C_1, C_2), (C_1, C_3), (C_2, C_3) \rangle$

**Step 4 (Calculate a closure)** *As long as the list of pairs is not empty, we perform a closure operation: For every two concepts,  $c_1 = \langle O_1, A_1 \rangle$  and  $c_2 = \langle O_2, A_2 \rangle$ , that do not dominate each other ( $c_1 \not\prec c_2$  and  $c_2 \not\prec c_1$ ), we take the set  $A_1 \cap A_2$  and calculate its set of common objects. Together, these two sets may represent a new concept. If that is indeed the case, we will add new pairs to the list for every other foreign concept.*

We begin with  $C_1$  and  $C_2$ . The intersection of their attributes is  $\{\text{Pnt}, \text{setXY}, \text{draw}\}$ , and the set of common objects of these attributes is  $\{x, y\}$ . Therefore,  $C_4 = \langle \{x, y\}, \{\text{Pnt}, \text{setXY}, \text{draw}\} \rangle$ . Since this concept dominates both  $C_1$  and  $C_2$ , we only add the pair  $(C_3, C_4)$  to the list of pairs. Next, we examine  $C_1$  and  $C_3$ . The intersection of their attributes is  $\{\text{Pnt}, \text{draw}\}$ , and the common objects of these is the set of all fields:  $\{x, y, \text{color}\}$ . Hence, we found our top concept,  $C_5 = \langle \{x, y, \text{color}\}, \{\text{Pnt}, \text{draw}\} \rangle$ , and cannot add new pairs to the list. Only two more pairs remain in the list:  $(C_2, C_3)$  and  $(C_3, C_4)$ . The closure of both pairs is  $C_5$ , which we already found. The list of pairs is now empty and we have all six concepts, which can be presented in a lattice as seen in Fig. 3.7.

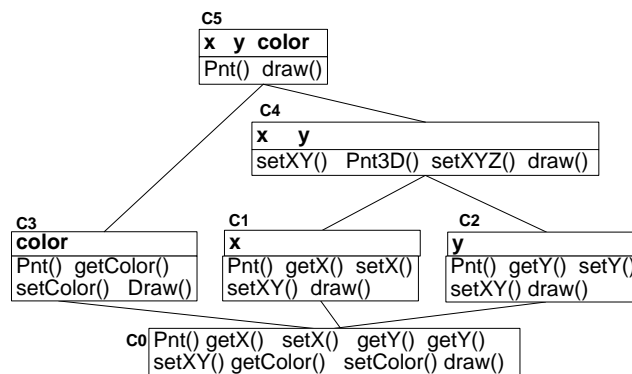


Figure 3.7: Fully annotated concept lattice of the Pnt class.

Once we have the fully annotated lattice, we can calculate the sparse annotations of the concepts.

# Chapter 4

## Concept Lattices of Classes

In the previous chapter, we saw a context of the accesses to the fields of a class and created the concept lattice of this context. In this chapter, we shall see how the concept lattice can help discover information about a class without reading its source code. Sec. 4.1 explains how to interpret this lattice and draw some conclusions from its structure. Sec. 4.2 discusses the removal of elements from the complete context and the creation of appropriate *subcontexts*.

### 4.1 Interpretation of concept lattices

If two methods are introduced in the same concept, their corresponding columns in the context table are identical. Hence, a concept lattice built upon the relation of accesses to fields partitions the methods of the class into *equivalence classes*. All the attributes introduced in each concept represent methods which access the same combination of fields. This combination is the set of fields in the full annotation of that concept. In other words, it is the union of the fields represented by the objects introduced in that concept and in every other concept that it dominates. For example, consider again the lattice of class `Pnt3D` which appears in Fig. 3.6. In this lattice, the `Pnt3D` and `draw` methods use all four fields while `setXY` uses only the `x` and `y` fields.

The structure and contents of a concept lattice allow us to make several deductions about the class it represents. These deductions can be made even in the absence of the original source code, making our methodology applicable for the task of reverse engineering classes.

We use the lattice in Fig. 3.6 to make the following deductions about the `Pnt3D` class:

1. *Every method uses at least one field, but no field is used by all the methods.* This deduction follows from the fact that the bottom concept ( $C_1$ ) is empty.
2. *There is a certain cohesion between all the fields of the class: all of them are required together to accomplish the drawing responsibility of the class.* We deduce this from the top concept (concept  $C_8$ ) which contains the nontrivial `draw` method.
3. *There is a lack of symmetry between the three coordinate components.* This asymmetry is visually evident from the existence of concept  $C_5$  and the lack of two complementing methods, `setYZ` and `setZX`. Even if we were not aware of the fact that `Pnt3D` inherits from `Pnt`, this asymmetry hints that field `z` might have been added to the class at a later stage of its evolution or via inheritance.

4. *The class has two parts. One part, consisting of concepts  $C_3$ – $C_7$ , is responsible for maintaining the three coordinates. The other part, consisting only of concept  $C_2$ , is responsible for maintaining color information.* These parts are obtained by removing the top concept and the bottom concept, resulting in two disjoint graph components. Lattices which can be broken into disjoint graphs in this manner are called *horizontally decomposable lattices*<sup>1</sup>.

In Chapter 5 we will explore horizontally decomposable lattices and their implications in depth. One obvious implication is that none of the methods in one component invokes methods or access fields in other components. Hence, a horizontally decomposable lattice can sometimes be used to detect a separation of concerns inside the class. The existence of such a separation may simplify the understanding of the class but might also indicate a problems in its cohesion.

In the case of the `Pnt3D` class, the existence of the constructor and the `draw` method that combine the two parts nullifies the possibility that the `Pnt3D` class has two independent responsibilities. It does, however, suggest a possible redesign of this class, as an aggregate of two new classes, `Coordinate3D` and `Color`.

For the lattice of the simple `Pnt3D` class, it is straightforward to examine the concepts and immediately draw conclusions. The lattices of large and complex classes, however, tend to be large and complex themselves, and therefore a more organized approach is needed to deal with them. Such an approach is suggested in the following chapters, in the form of our three-stage methodology.

In order to understand a concept lattice of a large class, it is often also helpful to first examine a simpler lattice based upon a subset of the complete context, which we call a *subcontext*. In fact, we have already worked with subcontexts: The *concrete context* is a subcontext of the complete context. We discuss subcontexts in depth in the next section.

## 4.2 Additional contexts

### 4.2.1 Subcontexts

Consider again the concrete context of class `Pnt3D`, which was depicted in Table 3.2. Suppose that we now create a new context which is identical to the original context in all columns and rows except for the removal of the column for the constructor, `Pnt3D()`. The sets of objects and attributes of the new context are now not necessarily strict subsets of those of the original. For every object-attribute combination in the new context, the corresponding table entry agrees with that of the original context. Hence, the new context is a *subcontext* of the complete context, or conversely, the complete context is a *supercontext* of the new one. More formally:

**Definition 10 (Subcontext)** *Let  $C = \langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$  and  $C' = \langle \mathbf{O}', \mathbf{A}', \mathbf{R}' \rangle$  be two contexts. We say that  $C'$  is a subcontext of  $C$  if and only if the following conditions hold:*

1.  $\mathbf{O}' \subseteq \mathbf{O}$ .
2.  $\mathbf{A}' \subseteq \mathbf{A}$ .
3.  $\forall (o, a) \in \mathbf{O}' \times \mathbf{A}' : (o, a) \in \mathbf{R}' \iff (o, a) \in \mathbf{R}$ .

---

<sup>1</sup>See Definition 13 in Chapter 5.

The lattice of the context resulting from removing only the constructor of `Pnt3D` is identical to the lattice of the complete context of that class, except that the annotation of the top concept no longer includes the constructor.

The structure of a lattice of a subcontext is not always identical to the lattice of the supercontext. To see this, consider a context  $C$  and a subcontext  $C'$ , and let  $o_1$  and  $o_2$  be two objects that exist in the sets of objects of both contexts (we can make the symmetrical arguments for attributes). If  $o_1$  and  $o_2$  appeared in the same concept of  $C$ , then they must also appear in the same concept of  $C'$ , because they share the same attributes in both contexts. However, even if  $o_1$  and  $o_2$  appeared in different concepts of  $C$ , they may still appear in the same concept of  $C'$  if the attributes in which they are different are removed in the subcontext. It follows that the lattice of a subcontext can be simpler than that of its supercontext, because some concepts can completely disappear, their remaining contents *merged* with those of other concepts.

For example, suppose that we remove the object representing field `z` from the concrete context of `Pnt3D`. The resulting lattice appears in Fig. 4.1. Methods `getZ` and `setZ` are merged into the bottom concept because they now use no fields, and their original concept,  $C_6$ , is removed. In addition, method `setXYZ` joins the concept of `setXY`,  $C_5$ , and its original concept,  $C_7$ , is removed.

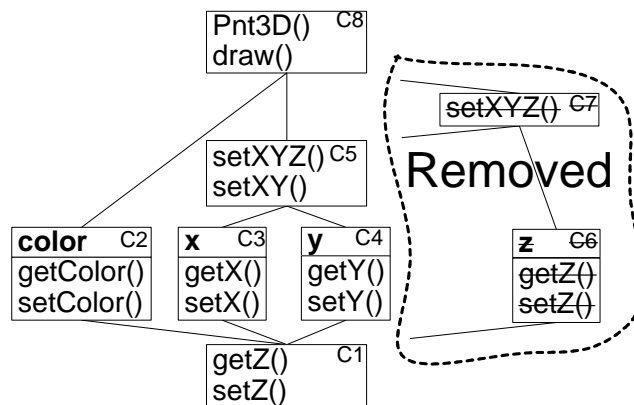


Figure 4.1: Concept lattice of class `Pnt3D` after field `z` is removed from the context.

The above example shows that we must carefully consider the kinds of methods, fields, and accesses that we remove from our context, or we will obtain a misleading lattice.

## 4.2.2 Selecting Methods

When creating a subcontext, we will usually be removing methods. Table 4.1 lists the different criteria available for selecting the methods which will take part in a context. The underlines in the table indicate the selections that represent the complete context.

Each criterion in Table 4.1 has either a *yes/no* choice or a *yes/no/don't care* choice. For a *yes/no* criterion ( $y \mid n$ ), an  $n$  selection means that no method with that property will appear in the context, regardless of other criteria. A  $y$  selection allows the method to appear in the context if it is not rejected due to other criteria. For a *yes/no/don't care* criterion ( $y \mid n \mid \phi$ ), a  $y$  selection rejects methods *without* the property, an  $n$  selection rejects methods *with* the property, and a  $\phi$  selection has no effect. Hence, the underlined selections in the table indicate that no method is rejected.

Criteria	Options
public	<u>y</u>   n
protected	<u>y</u>   n
package	<u>y</u>   n
private	<u>y</u>   n
static	y   n   <u>ϕ</u>
special	y   n   <u>ϕ</u>
inherited	y   n   <u>ϕ</u>
overridden	y   n   <u>ϕ</u>
effective	y   n   <u>ϕ</u>

Table 4.1: Method selection options  
(Underlines indicate the selections representing the complete context)

We now turn to discussing each criterion in depth.

### Member visibility

The complete context of a class includes all methods regardless of their declared level of visibility. The encapsulated methods (typically low-level) are required to fully-understand the implementation of the class, but can be ignored if we want to understand the higher level operations and in particular those which constitute the interface of the class.

The JAVA language supports four visibility levels:

**public** Visible to every client in every package. These methods are always part of the interface of the class.

**protected** Visible only to inheriting classes. These methods can be regarded as part of the interface presented to a client who derives from the current class.

**package (default)** Hidden from classes outside the package. These methods are part of the interface presented to clients in the same package.

**private** Not visible to anyone outside the class.

Depending on the stage of the class analysis, we shall select methods at different levels of visibility. Typically, this will involve choosing a minimal visibility level and keeping all the methods in- or above- that level<sup>2</sup>, such as keeping only **public** methods. There is no value in making an opposite selection, for example by keeping **private** methods while removing more visible ones.

### Static methods

Because every function in JAVA is a member of a class, programmers write utilities in the form of **static** methods. When we want to study how an instance of a class behaves, it often makes sense to remove such methods.

On the other hand, we may need to focus on these methods when studying *utility classes*, classes that group many **static** methods. If the methods in such classes make use of **static** fields, then FCA might help us classify them to groups of related functionality.

<sup>2</sup>Note that the visibility levels of **protected** members are not comparable with those of **package** members.

## Special methods

When we want to understand the specific functionality of the class, we are often not interested in methods whose functionality is not unique to the class or to the domain they appear in. The methods defined in `java.lang.Object` are available in every class and therefore always removed (unless overridden). In addition, we define a set of *special methods* which are not specific to the functionality of the class:

**Definition 11 (Special methods)** A special method is a constructor, a finalizer or one of the following: `clone`, `equals`, `compareTo`, `readObject`, and `writeObject`.

The above definition can be extended to other general operations in order to fit the system under analysis. For example, if all the classes in the application inherit from a single base class, it might be preferable to treat all the methods of that class as special.

Every method can be either special or non-special. A **y** selection in the *special* criterion selects only special methods, whereas **n** selects all other methods.

## Inherited methods

The set of methods of a particular class includes new methods defined (given a body) in that class, as well as methods inherited from the super classes. In studying a class which appears deep in the inheritance tree, it is sometimes useful to first consider only the inherited methods, then consider only the new methods, and finally examine both kinds together. Hence, a **y** selection in the *inherited* category will select only the inherited methods, whereas a **n** selection will select only the new methods. Note that overridden methods are still considered inherited. If we choose a “don’t care” for this criterion, we may end up with several methods with the same signature.

## Overridden methods

An overriding method *hides* all the overridden versions from clients. Therefore, it is often useful to reject all overridden methods from the context (an **n** selection), and keep only the most overriding version. A **y** selection will keep all the overridden versions and reject the overriding ones and is therefore not particularly useful. Note that we consider all the constructors of a class as overridden by all the constructors of the subclasses, regardless of the exact signature. We do so because only the constructors of the subclass are accessible to clients of that class; the constructors of the superclass may only be invoked from inside the subclass.

## Effective methods

If we want to fully understand the implementation of a class, we need to consider every method which might be executed. In JAVA, an overridden method can be invoked by the overriding method using the **super** keyword. As a result, removing all overridden methods from the context can hinder the analysis of the implementation of the class, but keeping all of them might clutter its lattice.

To alleviate this problem, we have the option of keeping only *effective methods*, which can actually be executed. We formally define these methods recursively, as follows:

**Definition 12 (Effective method)** Let  $C$  be a class, and let  $m$  be a method in the set of methods of  $C$ . **Base:** If  $m$  is defined (has a body) in  $C$  then  $m$  is effective in  $C$ . Otherwise, let  $C'$  be the superclass of  $C$  in which  $m$  is defined. If  $m$  is defined with **public** or **protected** access and  $m$  is not overridden in a class between  $C'$  and  $C$ , then  $m$  is effective in  $C$ . If  $m$  is defined with **package** access,  $C'$  resides in the same package as  $C$ , and  $m$  is not overridden in a class between  $C'$  and  $C$ , then  $m$  is effective in  $C$ . **Step:** Let  $m'$  be another method in the set of methods of  $C$ . If  $m'$  is effective in  $C$  and  $m'$  calls  $m$  directly, then  $m$  is also effective in  $C$ .

To remove only *ineffective* methods from the context and keep the effective ones, we must select  $\phi$  in the *overridden* criteria and  $y$  for *effective*.

The concrete context, which we first encountered in the previous chapter, is in fact based on these selections, as can be seen in Table 4.2.

Criteria	Options
public	<u>y</u>   n
protected	<u>y</u>   n
package	<u>y</u>   n
private	<u>y</u>   n
static	y   n   <u><math>\phi</math></u>
special	y   n   <u><math>\phi</math></u>
inherited	y   n   <u><math>\phi</math></u>
overridden	y   n   <u><math>\phi</math></u>
effective	<u>y</u>   n   <u><math>\phi</math></u>

Table 4.2: Method selection options for the concrete context  
(Underlines indicate the selections representing the concrete context)

### 4.2.3 Selecting Fields

All fields must be kept in the context regardless of the level of abstraction at which we analyze the class. The reason for this is that the fields are the basis upon which the lattice is constructed and the methods are partitioned. For example, suppose that in order to analyze the interface of a class we remove from the context not only methods which are not visible to clients, but also the unexposed fields. Since in a well-written class no field is exposed, the set of objects of the context is likely to be empty and the methods cannot be partitioned, resulting in a trivial *singleton lattice*. Another reason for striving to keep fields in the context is that their number is usually very low compared to the number of methods which operate upon them.

Nevertheless, after we have constructed the lattice, it is sometimes useful to remove fields with particular properties. For example, we may want to remove the encapsulated fields when analyzing the interface of a class, in order not to expose any implementation details. Table 4.3 summarizes the options for selecting the fields which would be displayed in the final lattice. The semantics are similar to those dealing with method selection, except that *ineffective fields* are encapsulated fields that are not accessed by any method.

Criteria	Options
public	<u>y</u>   n
protected	<u>y</u>   n
package	<u>y</u>   n
private	<u>y</u>   n
static	y   n   <u><math>\phi</math></u>
inherited	y   n   <u><math>\phi</math></u>
effective	y   n   <u><math>\phi</math></u>

Table 4.3: Field selection options  
(Underlines indicate the selections representing the complete context)

Note that the impact of **static** fields with **public** or default access on the class in which they appear is limited, as they often serve as application- or package- wide global variables. To gain more information from such fields, a system-wide concept analysis should be performed in a manner similar to that used in works which analyze the use of global variables by procedural code for automatic modularization (e.g. [48]).

#### 4.2.4 Selecting Accesses

In many cases, such as typical field-update scenarios, a method can access the same field multiple times and in different ways. Because the binary nature of formal contexts does not allow their entries to indicate the type or cardinality of the accesses, a predicate must be arbitrarily defined for what constitutes an access from a method to a field. In this work, an entry for a method and a field appears in the context if the method accesses the field at least once, regardless of whether the access is direct or indirect or whether it is a read or write access.

The definition of subcontexts requires that there will be an agreement between the table entries of the subcontext and the supercontext for pairs of objects and attributes that exist in both contexts. We define a *pseudo-subcontext* to be a subcontext in which this requirement is relaxed. In other words, a *pseudo-subcontext* can contain a subset of the entries of its supercontext.

Table 4.4 summarizes options for removing relation entries for accesses. An entry for a particular method and fields exists in the resulting context if the method accesses the field in at least one manner that is not filtered out by the selections in Table 4.4.

Criteria	Options
direct	y   n   <u><math>\phi</math></u>
indirect	y   n   <u><math>\phi</math></u>
reads	y   n   <u><math>\phi</math></u>
writes	y   n   <u><math>\phi</math></u>

Table 4.4: Access types selection options  
(Underlines indicate the selections representing the complete context)

In practice, useful contexts will not filter out methods simply because they perform a direct or an indirect access. As discussed in Chapter 2, we will rarely use only direct accesses because in well-

written class, fields are accessed only via simple *get* and *set* methods. On the other hand, filtering methods according to the kind of effect they have on a field is often useful because sometimes the operations which affect the state and those that do not affect it are orthogonal. In fact, the lattices of some classes can be greatly simplified by considering the lattices for read and write accesses separately.

We demonstrate such a case on the `sun.net.www.MimeEntry` class from version 1.4.1 of the JDK<sup>3</sup>. The concept lattice for this class, which appears in Fig. 4.2 is quite complicated, as there are five layers and also intricate connections between the concepts of the upper layers.

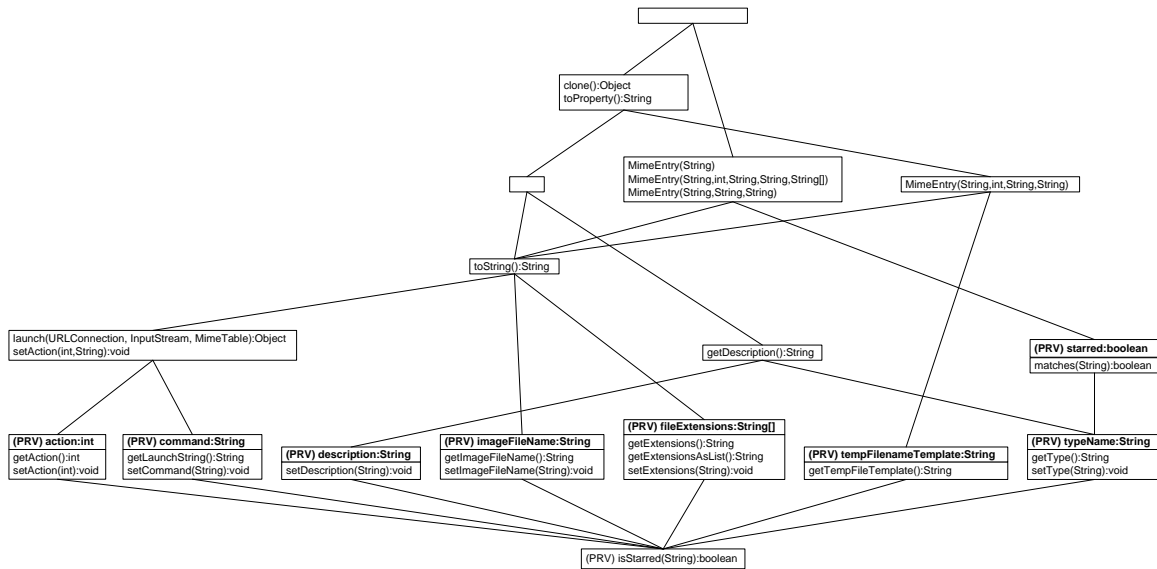


Figure 4.2: Concept lattice of all the accesses to fields in the `sun.net.www.MimeEntry` class.

An examination of the accesses table of the class, which appears in Table 4.5, shows that almost all the methods can be divided to methods that only read fields and to methods that only write fields. Furthermore, whereas small combinations of fields tend to be associated both with methods that read the fields and methods that write them, most of the larger combinations are used solely by one type of access. We therefore create a lattice for read accesses, depicted in Fig. 4.3, and a lattice for write accesses, depicted in Fig. 4.4.

The two lattices for the specific kinds of accesses sport less layers than those in the one that includes all accesses. Moreover, the number of crossing edges, which tends to confuse readers, is decreased.

In addition to all the selection criteria discussed above, it is possible to declare *threshold criteria* such as “select only methods that write to the same field twice”. However, such criteria are only useful in rare cases and therefore not discussed further.

<sup>3</sup>The class file for this class appears in the `dt.jar` archive file under `jre/lib`.

		methods							
		action:int	command:String	description:String	imageFileName:String	fileExtensions:String[]	starred:boolean	tempFileNameTemplate:String	typeName:String
fields	clone():Object	RW	RW	RW	RW	RW		RW	R
	getAction():int	R							
	getDescription():String			R					R
	getExtensions():String[]					R			
	getExtensionsAsList():String					R			
	getImageFileName():String				R				
	getLaunchString():String		R						
	getTempFileTemplate():String							R	
	getType():String								R
	isStarred(String):boolean								
	launch(...):Object	R	R						
	matches(String):boolean						R		R
	void setAction(int,String):void	W	W						
	setAction(int):void	W							
	setCommand(String):void		W						
	setDescription(String):void			W					
	setExtensions(String):void					W			
	setImageFileName(String):void				W				
	setType(String):void								W
	toProperty():String	R	R	R	R	R		R	R
	toString():String	R	R		R	R			R
MimeEntry(String)	W	W		W	W	W		W	
MimeEntry(Str,int,Str,Str,Str[])	W	W		W	W	W		W	
MimeEntry(Str,int,Str,Str)	W	W		W	W		W	W	
MimeEntry(Str,Str,Str)	W	W		W	W	W		W	

Table 4.5: Field accesses relation in the sun.net.www.MimeEntry class.

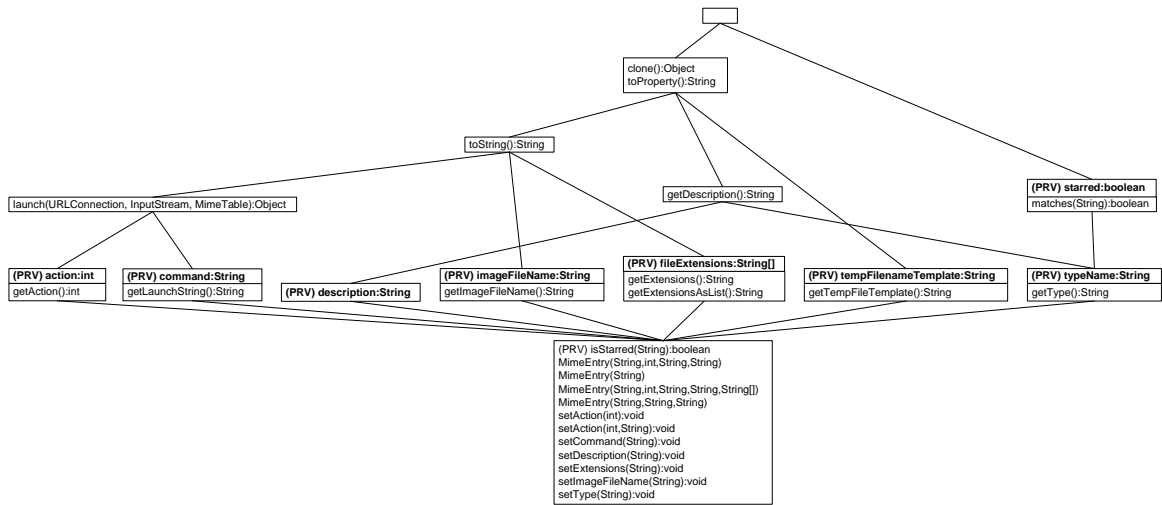


Figure 4.3: Concept lattice of read accesses to fields in the `MimeEntry` class.

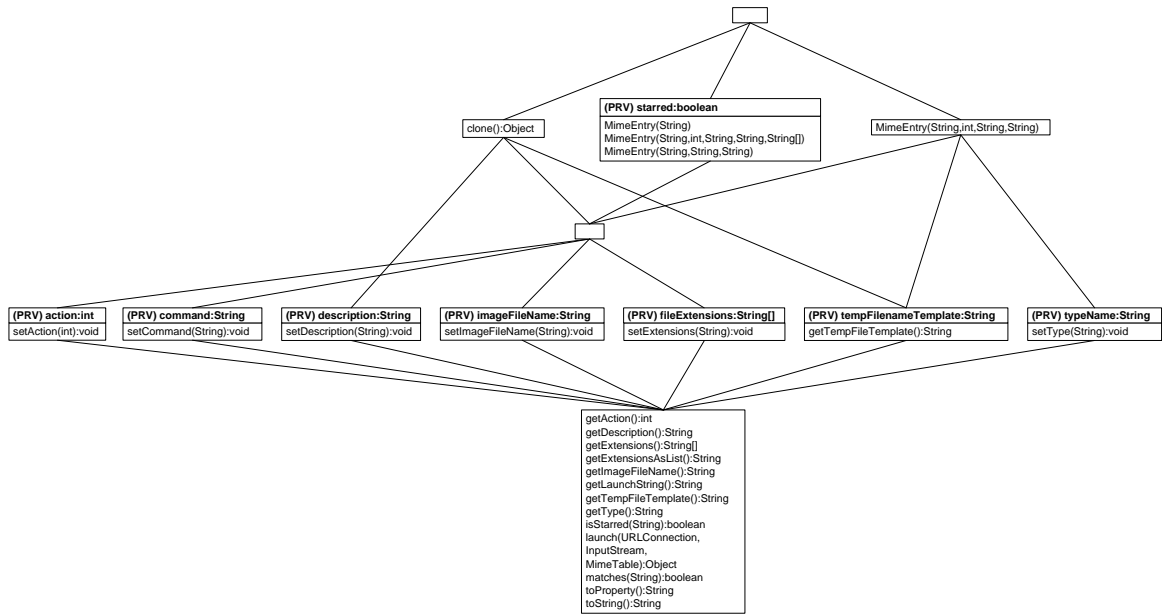


Figure 4.4: Concept lattice of write accesses to fields in the `MimeEntry` class.

# Chapter 5

## Methodology Stage I: Interface Analysis

### 5.1 Introduction

This chapter presents the first stage of our methodology, dealing with studying the interface of a class which typically consists of its **public** methods and fields. Many resources should be devoted to the maintenance of the interface because of its rigid nature (which makes changes costly and sometimes impossible) and since it is often the only concern of clients who wish to use the class as a “black box”.

To improve the chances of a class being effectively reused, its supplier must ensure that its interface is consistent and straightforward. A prospective client is likely to use a class only if it is reliable, supplies all the necessary functionality, and is easy to use. If the interface of the class is complex or un-intuitive, the effort required for its use increases and may become prohibitive. Unfortunately, classes that represent complex abstractions tend to sport a large and complex interface. This occurs especially if the *shopping-list approach* [42, pp.80–83] is used, because it increases the size of the interface without increasing the actual functionality provided by the class, and actually allows the same operation to be carried out in a number of ways.

The developers of a class with a large interface face a problem of maintenance since the evolution of the interface is difficult to track and maintain, and problems such as inconsistent-, redundant-, and obsolete- features are aggravated. As maintenance becomes complex, implementation details may proliferate into the interface and pose a problem for clients and developers. As we shall see in the examples provided in this work, implementation details often sneak into class interfaces even when the represented abstractions are relatively simple. Conversely, sometimes an interface is so un-intuitive or incomplete that implementation details must be explored before the class can be used as a black box.

In addition to keeping a large interface tidy, it is important to organize and present its members in a meaningful way. Standard documentation generation mechanisms, most notably JAVADOC [29], use a “trivial” ordering (e.g., lexicographic) for the methods. Meyer [42, pp.103–108] argues that an interface is better presented when it is *flattened* (with details of the underlying inheritance structure hidden) and organized by *feature categories*. However, unlike EIFFEL which supports a *flat-short form* [41, p.106], most languages do not support such a flattening, and few programmers provide an explicit categorization of the features in the classes they provide.

In this work we will use the lattices of classes obtained by FCA as a heuristics for automatically performing a sort of feature categorization. We argue that presenting the members of the class in such

a form helps clients and reviewers better understand the interface and discover problems.

This categorization is also the basis for our elaborate and structured class analysis methodology, whose first stage is presented in this chapter. This stage is restricted to the analysis of interfaces, and can be carried out even if no source code or additional documentation is available. In the next stages we will also examine the fields and the code of the class.

The remainder of this chapter presents nine steps or activities for the exploration and inspection of a class interface:

1. Set expectations.
2. Explore the environment of the class.
3. Select a context.
4. Lay out the concept lattice using *layers*.
5. Simplify concepts' annotations.
6. Perform *horizontal decomposition*.
7. Create an *abstraction lattice*.
8. Match services against expectations.
9. Identify core, auxiliary and wrapper services.

The above steps are not necessarily carried out in sequence, and some are further split into tasks. Each step is thoroughly described and demonstrated in the following sections, in a case study on the `Molecule` class from the CDK project.

## 5.2 The CDK case study

The *Chemistry Development Kit* (CDK) [10,52] is an open-source library of JAVA classes for cheminformatics and computational chemistry, that serve as a basis for other applications, such as *JChemPaint* [31], *JMol* [32], and *Seneca* [45].

Prior to the selection of this case study, we were not familiar with the library or affiliated with its authors in any way; nor did we have any particular knowledge of the application domain. An evidence to the efficacy of our methodology is that despite these limitations, we revealed problems which were confirmed as new errors by the developers and will be fixed in subsequent versions.<sup>1</sup>

Our case-study focuses on a single class named `Molecule` in the root package of the library. We selected this class because it represents an entity that should be familiar to most readers, and yet sports a very large interface consisting of 77 **public** members. The `Molecule` class itself derives from a class named `AtomContainer`, which in turn derives from a class named `ChemObject`. In accordance with Meyer's notion of a *flat-short form* [41, p.106], no distinction will initially be made in the lattices between members originating in different superclasses.

## 5.3 Step 1: Set expectations

Before approaching the actual class, we need to prepare ourselves and know what to expect. Having an estimate of the purpose, features, and terminology of the class in advance simplifies our first encounter

---

<sup>1</sup>Our case study dealt with build 20020518 of the library.

with it. It allows us to become familiar with the vocabulary and the “human context” at which the class operates. This familiarity is important because our main clues in studying the interface will be the names and signatures of methods. Having prior expectations will also improve our prospects of noticing problems such as missing or superfluous features, inconsistent interfaces, and exposed implementation details.

In addition, expectations about functionality allow us to approach the problem of *concept assignment*. In the literature, the problems of *concept assignment* [6, 7] or *feature-component mapping* [17] (with a different meaning to the word “concept” from its FCA meanings) refer to the determination of the program entities which provide each *feature* (a realized requirement or behavior of the program). In other words, we can think of these problems as mapping “human concepts” into corresponding program elements. Clearly, addressing these problems constitutes an important part of the understanding of software. Yet, the problem faced by a client who studies an unfamiliar class is even more complicated, because in addition to not being familiar with the program entities, this client does not know the exact specifications and requirements to which the class conforms. Hence, before the class itself is examined, an important preparatory step is to surmise at least a rough idea of the expected features.

In this step and in the next one, we use our knowledge about the problem domain and the program-environment in which the class resides to try and gather some expectations, without performing a detailed analysis of the actual source code or JAVADOC documentation. These steps have nothing to do with concept analysis and are not unique or new, and are therefore discussed and demonstrated briefly here.

### **Surmise the purpose and roles of the class.**

Based on its name and any additional information we have on the domain, we need to surmise the purpose of the class and estimate the roles in which it can serve. Determining the purpose of the `Molecule` class is relatively straightforward as it corresponds to a familiar real-world entity. Since a molecule is a chemical entity comprising of atoms which are connected by chemical bonds, we surmise that the primary responsibility of the class is to manage collections of atoms and bonds. We expect to encounter these terms in the class interface.

Without being familiar with the actual system in which the `Molecule` class is used, we can only speculate that the roles of its instances may include being members in some collection (such as a catalogue of molecules), or interacting with other `Molecule` instances in chemical reactions.

### **Delineate the responsibility of the class.**

In surmising the purpose of a class, it is important to try and delineate the limits of its responsibility. In other words, we should try and discover what the class is not. Otherwise, we may wrongly anticipate something which is too general or too specific.

In our example, we guess that although a molecule can probably consist of a single atom or a single pair of atoms connected by a single bond, the class library is likely to include other classes for representing these entities. Also, we guess that the class is not intended to simply represent two collections (or just one of them). We surmise this not only because a molecule should have additional responsibilities (which arise from its correspondence to a real-world entity) but from some additional knowledge on the domain: We know that other entities, such as mixtures, ions, and solutions, also

consist of multiple atoms and possibly some bonds, but are not considered as molecules. It is therefore possible that `Molecule` would be a subclass or aggregate of classes that abstract atoms and bonds, or collections of these.

Conversely, we expect `Molecule` to provide only features which are useful for all kinds of molecules, and not for specific types. More domain knowledge tells us that many different families of molecules with special properties exist, such as organic compounds or polymers. It is likely that if supported by CDK, these entities are realized as subclasses of `Molecule`. We will be able to check this assumption in the next step, when we explore the environment of the class.

### **Determine a vocabulary.**

In order to perform the concept assignment, we need to anticipate the terms we will encounter, and create a vocabulary of the problem and program domain, with a list of synonyms. We are bound to see the terms *atom* and *bond*, because their manipulation constitutes the primary purpose of the class. However, a bond might also be called a *connection*, or qualified as a *chemical bond* or *covalent bond*. Dealing with bonds may also lead to the use of related terms such as *electrons*, which are shared in a chemical bond. It might also involve a term that represents the number of shared electrons. Without being more familiar with the problem domain, we cannot predict the exact term for this notion, but we can expect it to indicate a magnitude and can therefore think of different candidates: *cardinality*, *magnitude*, *order*, *strength*, and *degree*.

Depending on the amount of functionality provided by the `Molecule` class, we may also encounter other chemical terms which can relate to molecules. For example, if the class supports the maintenance or calculation of chemical and physical properties of molecules, we might encounter terms such as *polarity*, *mass* and *temperature*.

### **Determine the functionality that the class must provide.**

Since we surmised that the primary purpose of the `Molecule` class is to maintain and manipulate collections of atoms and bonds, it follows that the class interface must include operations that provide this functionality. Namely, we should be able to add, access, and remove individual atoms and bonds. We should also be able to operate on multiple elements, by clearing, combining, or enumerating them. A class of this magnitude should also support standard operations like copying, cloning and serializing.

### **Speculate on additional functionality which the class may provide.**

The `Molecule` class might provide additional features that are not directly related to atoms and bonds. For example, it might support naming or cataloguing the molecules, perhaps automatically using a standard nomenclature. The class might even provide operations for calculating chemical or physical properties, such as its polarity or mass.

Note that it is not necessary to guess in advance all the functionality which the class might provide. We might not even be able to do so without additional domain-specific information and more knowledge about the requirements of the system. We can comprehend a class even if we do not guess any additional functionality, but our task is easier if we do.

## 5.4 Step 2: Explore the environment of the class

The environment of the class under inspection, namely the other classes which reside in its directory and especially those upon which it depends, can provide valuable information that can help us guess additional functionality and validate or refine our earlier assumptions. A JAVA class *depends* on another class if it subclasses it, instantiates it, or references it in its code. Automatic software tools can help us identify the dependencies without examining the source code ourselves. We may also refer to the JAVADOC descriptions of the classes in the systems.

### Examine the interfaces implemented by the class

We examine the interfaces that a class implements to gain further insight into its domain and functionality. These interfaces may tell us something about the responsibilities assumed by the class, and assist in the categorization of its methods. Unfortunately, the interfaces `Cloneable` and `CDKConstants` which are implemented by `Molecule` (via its parents) do not tell us much about the class. The first is pretty standard, while the second turns out to be a programmatically questionable way to realize package-wise global constants.

### Examine super- and sub- classes

Familiarity with the location of the class under inspection in the hierarchy of classes yields substantial information that helps us better identify and delineate the purpose and roles of the class.

Consider Fig. 5.1, which depicts the hierarchy of classes in the root package of CDK. For now we ignore the various problems in the hierarchy, and focus on the central branch of the figure, where `Molecule` is located.

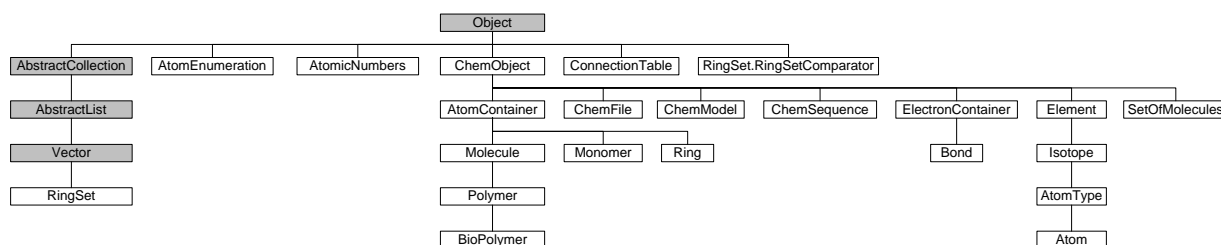


Figure 5.1: Hierarchy of classes in the root package of CDK  
(Shaded boxes represent classes outside the package)

According to Fig. 5.1, `Molecule` inherits from `AtomContainer`, which in turn subclasses `ChemObject`. The latter seems to serve as a base for many other classes in the package. We also see that the only subclass (in this package) of `Molecule` is `Polymer`, which is further subclassed by `BioPolymer`.

From its name and the structure of the hierarchy, we surmise that `ChemObject` serves (or is at least supposed to serve<sup>2</sup>) as the base class of all the classes for which there are corresponding chemical entities in the real world. In other words, it serves as a root for classes having to do with the problem

<sup>2</sup>There are apparently problems with classes `ChemFile` and `SetOfMolecules`, but they are ignored since we are currently interested only in what affects the `Molecule` class.

domain. The other classes in the package, which inherit from `Object` directly, appear to relate to program-space entities, such as collections and enumerations.

To learn more about the purpose of `ChemObject` we examine the associated JAVADOC comment, which describes it as “*The base class for all chemical objects in CDK. It provides methods for adding listeners and for notification of events, as well hash tables for manipulation of physical or chemical properties*”<sup>3</sup>. Hence, the management of properties, which we thought of as a feature potentially provided by `Molecule`, is provided by all the chemical entities. The mentioning of a hash table suggests that these entities (including `Molecule`) settle for collection-based manipulation of properties and do not provide functionality for their automatic calculation. This nullifies one of the expectations we set in the previous step.

Next, we examine `AtomContainer`. At a first glance, the existence of a base class for the management of atoms, as well as the number of classes that extend it, appears to confirm our premise that `Molecule` is not the only collection of atoms in the system. It leads us to assume that `Molecule` is different from `AtomContainer` in that it also includes a collection of bonds. However, an examination of the JAVADOC description of `AtomContainer` negates this possibility, and shows that the name given to class `AtomContainer` is wrong. The class is described as a “*Base class for all chemical objects that maintain a list of atoms and bonds*”. The important consequence of this new information is that the primary functionality of `Molecule` is actually provided by `AtomContainer`. We can only assume that `Molecule` is different from `AtomContainer` in that it supports special properties and features of molecules which are not covered by the physical and chemical properties maintained by `ChemObject`.

We now turn to the subclasses of `Molecule`, which (in the root package) appear to be concerned with polymers. A glance at the JAVADOC comments for these classes confirms that they add family-specific information, confirming our earlier predictions.

### Investigate types used by the class interface.

The types which appear in the interface of a class can introduce new vocabulary and suggest additional functionality. The use of classes from the same program also serves to highlight strong dependencies between classes. In JAVA, this information can be automatically collected using classfile analyzers or utilizing the reflection mechanism.

We can separate the classes to which the interface refers into two groups: domain-specific classes, and standard library classes. As expected, the `Molecule` class uses classes named `Atom` and `Bond` in many of its methods, affirming our hypothesis about the existence of separate classes for these entities. It also refers to `ChemObjectListener`, which (according to its name) obviously implements a listener based on the *Model/View/Client model* [34].

From the standard library, `Molecule` refers to common types like `String` and `Vector`. Surprisingly, the `Point2D` and `Point3D` classes from the graphical AWT toolkit are also used. This new information suggests that `Molecule` might provide the capability of calculating the spatial properties of a molecule or even of individual atoms inside it.

In addition to the classes referenced in the arguments and return types of methods, the exception classes that the method can throw (declared using the `throws` construct) are also interesting. We

---

<sup>3</sup>The original text contained some grammatical errors which were fixed in this and in other quotations.

learn that there are exception classes for atoms (e.g. `NoSuchAtomException`) but no corresponding classes for bonds.

### Explore types used in method implementations and in hidden members.

Although more volatile than the types used in the interface, the types used in the implementations of methods can sometimes provide information that can help in comprehending or in validating earlier assumptions about the class. For instance, if a method delegates to another class, the name of the other class can sometimes suggest a new term or feature which we did not anticipate.

In the case of `Molecule`, the code of one of the methods refers to `AtomEnumeration`. This strengthens our assumption that there is an operation for enumerating the atoms in the molecule. Surprisingly, there is no corresponding class for the bonds.

### Explore other classes in the library

We already examined the classes upon which `Molecule` depends, but other classes in its library may also provide information. For example, according to the class hierarchy in Fig. 5.1, the root package of CDK contains a class named `SetOfMolecules`. The existence of a homogenous collection of molecules confirms our earlier assumption that one of the roles of `Molecule` is to serve as a member of a collection.

## 5.5 Step 3: Select a context

Before a lattice is constructed, an appropriate class context must be selected, based on the criteria presented in Chapter 4. The selected context should contain only members which are exposed as an interface to the particular client, so that implementation details are not exposed. For the purpose of our discussion here, we will assume that the prospective client of the class under analysis is an unrelated class in another package and hence use only **public** methods. For inheriting clients or clients in the same package, the selection is naturally different.

The inheritance structure of the class is suppressed because we are not interested in any implementation details while analyzing the interface. We need something similar to what is called in EIFFEL jargon the *flat-short form* [41, p.106]. All non-overridden inherited methods are included in the context, but the labels do not indicate the defining class. Also, for the initial examination of the class, the constructors and other special methods are kept.

In order to correctly classify the methods by use of fields, we will include all fields in the context table, but remove all the hidden ones from the resulting lattice. We will use all accesses to fields, regardless of their read/write or direct/indirect natures.

Fig. 5.2 summarizes the options for creating a context, and highlights the choices taken in our example. The semantics of the selections are the same as those discussed in Chapter 4.

The application of FCA to the context of `Molecule` described above conveniently organizes the 75 **public** methods and two **public** fields of the class in 24 non-empty concepts. As discussed in the next section, the selection of a particular layout in a lattice of this magnitude can affect the ability of readers to infer information from the lattice.

Methods		Fields (in lattice)		Access	
Criteria	Options	Criteria	Options	Criteria	Options
public	y   <u>n</u>	public	y   <u>n</u>	direct	y   n   <u>φ</u>
protected	y   <u>n</u>	protected	y   <u>n</u>	indirect	y   n   <u>φ</u>
package	y   <u>n</u>	package	y   <u>n</u>	reads	y   n   <u>φ</u>
private	y   <u>n</u>	private	y   <u>n</u>	writes	y   n   <u>φ</u>
static	y   n   <u>φ</u>	static	y   n   <u>φ</u>		
special	y   n   <u>φ</u>	inherited	y   n   <u>φ</u>		
inherited	y   n   <u>φ</u>	effective	y   n   <u>φ</u>		
overridden	y   <u>n</u>   <u>φ</u>				
effective	y   n   <u>φ</u>				

Figure 5.2: Summary of context selection decisions  
(Underlines indicate the default choices for interface analysis)

## 5.6 Step 4: Lay-out the concept lattice using layers

Consider again Fig. 3.6 from Chapter 3. Each of the concepts  $C_2$ ,  $C_3$ ,  $C_4$ , and  $C_6$  dominates the bottom concept directly. They are also similar in that each introduces a single field and an accessor and a mutator for that field. This similarity arises from the fact that the methods of each concept manipulate a single field, and operate at the same level of abstraction.

We refer to groups of concepts with similar topological properties as *layers*. In many classes, we found after dividing the concepts of the lattice into layers that concepts of the same layer have similar semantic properties, just like the concepts in Fig. 3.6 do. This finding concurs with our expectations that more sophisticated methods use more fields, and hence appear in higher layers in the lattice.

The use of layers in the layout of concept lattices or directed graphs is not new. Layers are used by graph-layout algorithms to obtain a meaningful and visually pleasing layouts of concept lattices and to highlight meaningful substructures (e.g. [49,53]). In this work we use a slightly different definition of the layers, which is customized for lattices of the access relation between methods and fields.

Fig. 5.3 shows a partitioning of an example lattice into layers, based on the recursive definitions which follow. These definitions use  $C(c)$  to denote the set of concepts which are directly dominated by some concept,  $c$ , and  $P(c)$  to denote the set of concepts which directly dominate  $c$ .

Formally, the *bottom string* (also known as *bottom chain* [21]) consists of the bottom concept and every concept  $c$  such that  $P(c') = \{c\}$  for a  $c'$  which is in the bottom string. Similarly, the *top string* consists of the top concept and every concept  $c$  such that  $C(c') = \{c\}$  for a  $c'$  in the top string.

A concept  $c$  is in the *bottom layer*, also called *layer 1*, if  $c$  is not in the bottom string, but every concept that it dominates is. The *top layer* is defined symmetrically, except that a concept which can belong to both the bottom- and top- layers is arbitrarily assigned to the bottom layer.

All other concepts are in *internal layers*, which are defined recursively so that a concept  $c$  belongs to layer  $i$  if it **(i)** does not belong to the top layer, **(ii)** dominates only concepts in layer  $i - 1$  and below, and **(iii)** dominates at least one concept in layer  $i - 1$ .

The above definition of layers was used for laying out the concept lattice of class `Molecule`, which is depicted in Fig. 5.4. The figure includes the full signature of all 77 interface members, with the details of non-`public` fields hidden to prevent implementation details from being exposed.



As cluttered as Fig. 5.4 is, the layout highlights the fact that about half (14/26) of the concepts are in the bottom layer, i.e., represent basic operations such as inspectors, mutators, accessors, or delegators on minimal sets of fields<sup>4</sup>. Also visible are the “large” concepts,  $C_{17}$ ,  $C_{16}$ ,  $C_8$  and  $C_{23}$  (in a descending order of magnitude).

Note that the empty concept in the second layer ( $C_{26}$ ) can be replaced by additional edges that connect every directly dominating concept ( $C_{18}, C_{22}$ ) with every directly dominated concept ( $C_{11}, C_{12}$ ). The empty concept in the bottom layer ( $C_{12}$ ) represents a hidden field with no associated methods and cannot be replaced. In this work we choose to keep empty concepts in order to reduce the number of edges.

## 5.7 Step 5: Simplify concepts’ annotations

The main reason that Fig. 5.4 is so cluttered is that it lists all the members in each concept, along with their full signatures. To simplify the lattice, we are going to try and simplify the labels of each concept by summarizing the details of the members it introduces. This process cannot be performed automatically, but since it is based only on the signatures it can be done rapidly by a human without too much effort. Note that this process has no effect on the topological structure of the lattice or on the number of concepts. We later discuss how the actual structure of the lattice can be simplified.

**Create the summarized lattice by replacing the list of methods in the label of each concept with a more concise, semantical description of its role.**

We rely on the vocabulary and on the information gathered in the first two steps, and use the signatures of the methods to surmise their roles. Whenever possible, we should try and summarize the roles of groups of related methods. Methods with unknown terms and methods which could not be summarized as part of groups should be retained for further exploration.

For example,  $C_{16}$  (the second largest concept) has 8 methods: `contains(Atom)`, `get2DCenter`, `get3DCenter`, `getAtoms`, `getAtomNumber`, `getLastAtom`, `removeAtom(Atom)`, and `removeAtom(int)`.<sup>5</sup> The responsibilities captured by this concept can be summarized as *retrieving and removing atoms, retrieving the serial number of an atom, and calculating centerpoints*. Note that the exact meaning of center-points is unknown at this stage and is therefore kept in the description.

The methods of  $C_{21}$  are `toString`, `remove(AtomContainer)`, `removeAtomAndConnectedBonds`, and `getConnectionMatrix`. Although it is pretty easy to guess the meaning of the first three methods, `getConnectionMatrix` presents an unfamiliar term, the *connection matrix*, which we retain for later exploration. Similarly,  $C_6$  includes the `getCasRN` and `setCasRN` methods, telling us that the responsibility of this concept is to manage a “casRN” property, but the nature of this property remains unknown at this stage.

We refer to the lattice in which concept responsibilities are summarized as a *summary lattice*. The summary lattice of `Molecule` is depicted in Fig. 5.5.

---

<sup>4</sup>The existence of a concept in the bottom layer indicates that this concept introduces at least one new field, because otherwise it would not dominate the bottom concept.

<sup>5</sup>Here and henceforth, the full signature is trimmed down to the extent required to distinguish between overloaded methods.

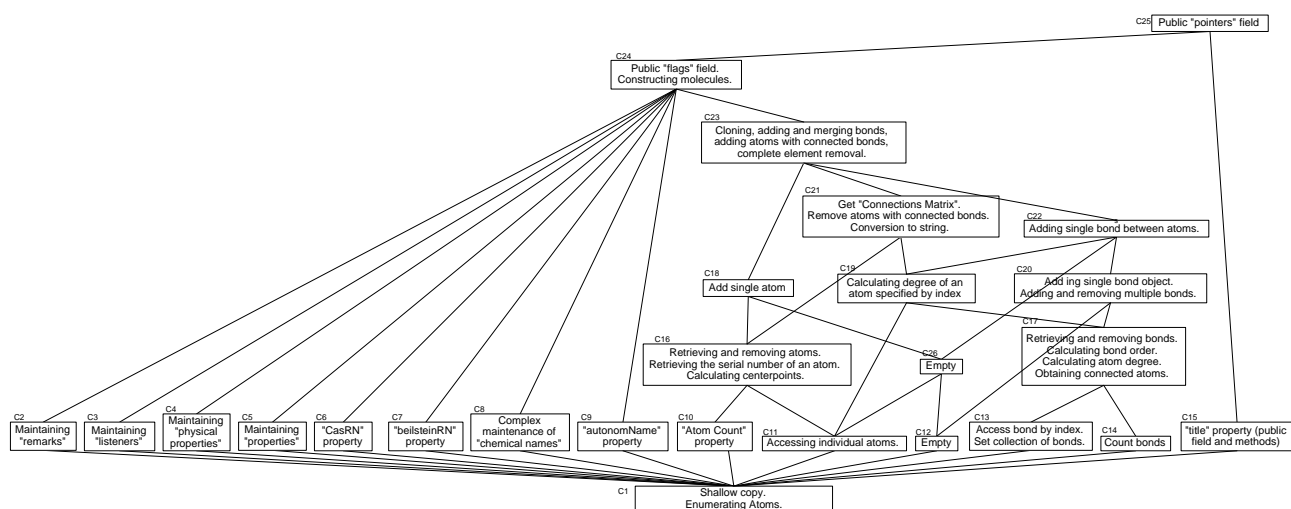


Figure 5.5: Summary concept lattice of `Molecule` with no fields

### Create the Outline lattice by giving short names to each concept

The labels of the concepts in Fig. 5.5 are still too verbose to allow a reader to understand each at a glance and comprehend the entire lattice. We need to replace them with clear short names. For this purpose, we introduce a naming convention for responsibilities and functionalities, which we call *functionality annotations*. The basic annotations for functionalities are listed in Table 5.1; they can be composed using the operators defined in Table 5.2

Symbol	Meaning
$\_x$	An exposed (e.g. <b>public</b> ) field named $x$ .
$@x$	Operate on a single $x$ element (read and write).
$\{x\}$	Operate on a single $x$ in a collection.
$\{\{x\}\}$	Operate on multiple $x$ 's in a collection.
$\{i \mapsto x\}$	Operates on a single $x$ in a collection that maps $i$ to $x$ .
$(i \mapsto x)$	Operates on a single $x$ via an $i$ (e.g. $x$ is a property or field of $i$ ).
$\#x$	Get an enumeration or an iterator of a collection of $x$ .
$ x $	Count elements in a collection of $x$ .
<i>cons</i>	Constructor.
" $x$ "	$x$ is an unfamiliar name.
<i>text</i>	free-text term.
? <i>text</i>	Boolean predicate on text.

Table 5.1: Basic *functionality annotations* for naming concept responsibilities

Using the convention of functionality annotations, we create the *outline lattice* of `Molecule`, which is depicted in Fig. 5.6. For example, concept  $C_{23}$  is given the name *clone | merge | intersect | clear*.

Guided by the layers and the newly found concept names, an examination of Fig. 5.6 reveals that the interface of `Molecule` can be divided into four main categories:

1. *Management of (nearly) the entire state*, as done in concepts  $C_{24}$  and (possibly)  $C_{25}$ .

Symbol	Meaning
$Rf$	Read-only/get versions of the functionality denoted by $f$ (if applicable).
$Wf$	Write-only/put versions of the functionality denoted by $f$ (if applicable).
$+f$	Add-only version of the functionality denoted by $f$ (if applicable).
$-f$	remove-only version of the functionality denoted by $f$ (if applicable).
$f_1 f_2$	Connects unrelated functionalities $f_1$ and $f_2$ in concept.
$f : x, y$	Functionality $f$ applies to both $x$ and $y$ .

Table 5.2: Composite *functionality annotations* for naming concept responsibilities

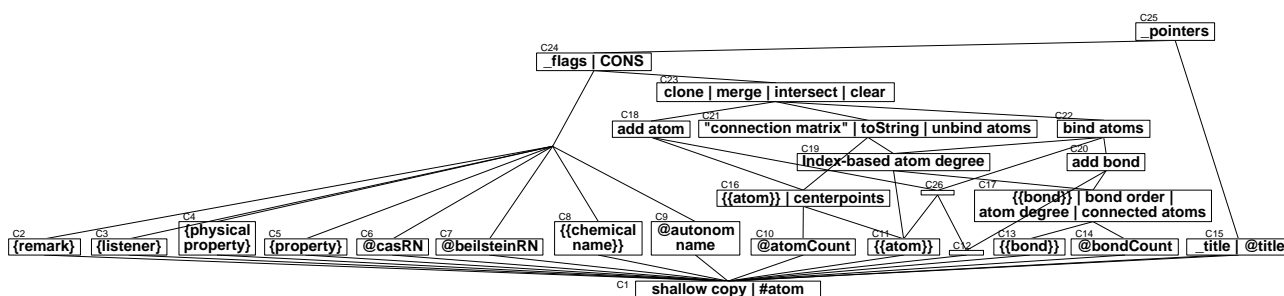


Figure 5.6: Outline concept lattice of `Molecule` with no fields

2. *Management of a large number of almost-independent properties in a record like manner.* Such properties include collections of *remarks*, *listeners*, *physical-* and *non-physical properties*, as well as two mysterious *casRN* and *beilsteinRN* properties, lists of *autonom-* and *chemical-* names, and a *title*. We infer that these properties are independent since there are no methods which combine them, except for those in the first category. We can see these methods in concepts  $C_2$ – $C_9$  and  $C_{15}$ .
3. *Direct management of interdependent properties.* These features include `atomCount`, `atom`, `bond` and `bondCount` in concepts  $C_8$ – $C_{14}$ . Their interdependency is revealed by the fact that they are dominated by concepts in the second and higher layers.
4. *Other methods dealing with abstractions of ties between atoms and bond.* These appear in concepts  $C_{16}$ – $C_{23}$  and  $C_{26}$ .

In the next step, we demonstrate how this breakdown can be supported by an automatic process of *horizontal decomposition*. Each of the “singleton components” corresponds to a property in the second category. Furthermore, the process highlights bugs and inconsistencies in the implementation.

## 5.8 Step 6: Perform horizontal decomposition

Consider again the concept lattice of class `Pnt3D` in Fig. 3.6. If the top- and bottom- concepts of the lattice are removed, we obtain two disjoint graph components, one dealing with coordinates and the other with color. These components suggest a restructuring of the class as an aggregate of two classes, `Coordinate` and `Color`.

Such a lattice, which consists of disjoint graph components that are connected only by the top- and bottom- concepts, is called a *horizontally decomposable* (HD) lattice:

**Definition 13 (Horizontally decomposable lattice)** Let  $G$  be the undirected graph obtained from a concept lattice  $L$  by ignoring edge directionality and removing the top- and bottom-strings. If  $G$  is unconnected, then we say that  $L$  is horizontally decomposable into components (or horizontal summands), each corresponding to a connected components of  $G$ . Singleton components are also called trivial components.

Efficient decomposition algorithms and a more formal treatment of horizontal decomposition can be found in [21].

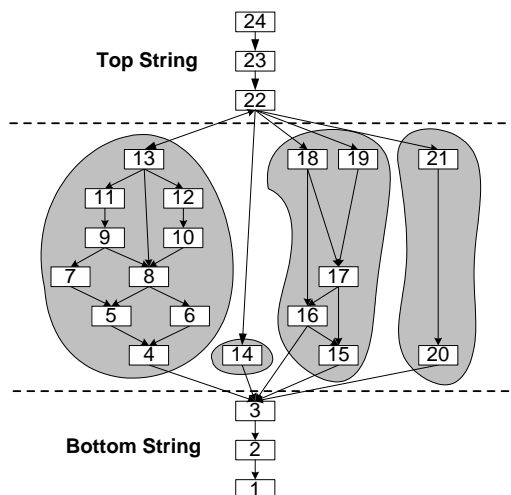


Figure 5.7: Horizontal decomposition of an example lattice

We apply the above definition to the concept lattice from Fig. 5.3 and discover that it is horizontally decomposable into one trivial component and three nontrivial ones, which appear shaded in Fig. 5.7.

The significant implication of horizontally decomposable lattices of classes is that *methods in one component cannot invoke methods or access fields in other components*. Thus, each component represents an independent functionality offered by the class. These functionalities are combined (if at all) only in high-level operations.

We now try to horizontally decompose the concept lattice of `Molecule`, which was depicted in Fig. 5.6. If  $C_{25}$ , the top concept, and  $C_1$ , the bottom concept, are removed from this lattice, we obtain the two disjoint graph components that are surrounded by thick dashed lines in Fig. 5.8.

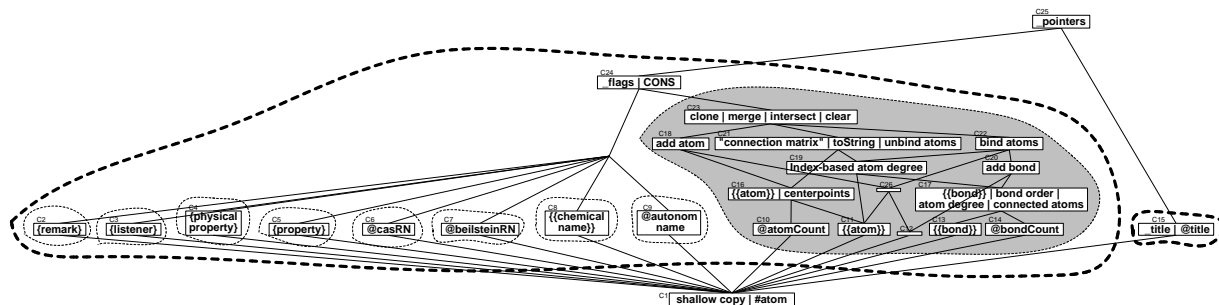


Figure 5.8: Horizontal decomposition of the outline concept lattice of `Molecule`

The rightmost component in Fig. 5.8 is trivial and consists only of concept  $C_{15}$ . The sole responsibility of this concept is to manage a `title` property. Even without delving into the details of the implementation, this horizontal decomposition highlights a potential problem: `title` is not handled by the constructor and the cloning operation, which respectively appear in concepts  $C_{23}$  and  $C_{24}$  of the other component. This problem was indeed confirmed as a new bug by the developers, to be corrected in a subsequent release. Another (unrelated) glitch is that the field itself is **public** even though it has an inspector and a mutator. Note that the `pointers` field (in  $C_{25}$ ) is also not used in construction and cloning.

If we treat the leftmost component (along with the bottom concept,  $C_1$ ) in Fig. 5.8 as a lattice, we can further horizontally decompose it into eight trivial components, surrounded by a thin dashed line in this figure, and one large nontrivial component, which is shaded. The trivial components correspond to the independent features of the class; each such component introduces an auxiliary field and several methods to manage it. Again, there is a potential problem with these fields because the cloning operation appears in the nontrivial component, to which we shall refer from now on as *component L*.

## 5.9 Step 7: Create an abstraction lattice

From the structure of the lattice of `Molecule`, it is clear that component  $L$  represents a more cohesive portion of the interface, which has to do with atoms, bonds and their interrelationship. However, the significance of each of the 14 concepts and 20 direct-dominance relations in it is not immediately obvious. In general, the outline lattice may still present too much information, which needs to be abstracted further.

We use methods of the top layer to group together concepts at lower layers. The rationale is that these methods use the largest subsets of fields and represent the highest level of abstraction. If two fields (or sets of fields) are always used together in higher abstractions, then we are inclined to believe that there is a strong tie between the two sets, and that the differences between them are due to low-level implementation details.

The abstraction is performed by creating a new concept lattice, which we call an *abstraction lattice*, based on the original one. The abstraction lattice is constructed for the *abstraction context*, where the objects are the concepts of the original lattice, the attributes are the concepts of its top layer, and an object has an attribute whenever there is a dominance relationship between the corresponding concepts. More formally:

**Definition 14 (Abstraction context)** Let  $G = \langle V, E \rangle$  denote the graph of a concept lattice, where  $V$  represents the entire set of vertices (concepts) of  $G$  and  $V_{top}$  represents the set of concepts constituting the top layer of  $G$ . We define the abstraction context of  $G$  to be the context  $\langle V, V_{top}, R \rangle$ , where  $(v, v') \in R$  if and only if  $v' = v$  or  $v'$  dominates  $v$ .

We refer to each concept in the abstraction lattice as a *cluster* because it essentially clusters all the concepts of the original lattice which it introduces as objects.

If a lattice is horizontally decomposable, its abstraction lattice will have a cluster for each component and not yield any useful information. For this reason, we do not apply this methodology to

the entire lattice of `Molecule`. Instead, we apply it to component  $L$  in the hope of simplifying it.<sup>6</sup> Table 5.3 lists the abstraction context of component  $L$ , and Fig. 5.9 depicts the resulting abstraction lattice, superimposed on the original lattice. The names of the clusters were chosen manually.

		attributes		
		$C_{20}$	$C_{21}$	$C_{22}$
objects	$C_1$	✓	✓	✓
	$C_{10}$	✓	✓	
	$C_{11}$	✓	✓	✓
	$C_{12}$	✓		✓
	$C_{13}$		✓	✓
	$C_{14}$		✓	✓
	$C_{16}$	✓	✓	
	$C_{17}$		✓	✓
	$C_{18}$	✓		
	$C_{19}$		✓	✓
	$C_{20}$			✓
	$C_{21}$		✓	
	$C_{22}$			✓
	$C_{23}$			
$C_{26}$	✓		✓	

Table 5.3: Abstraction context of component  $L$  from the concept lattice of `Molecule`

The abstraction lattice of component  $L$  in Fig. 5.9 organizes the 14 original concepts (15 with  $C_1$ ) while reducing the number of edges from 20 (25 with  $C_1$ ) to 10.

Note that an edge between clusters indicates that at least one concept (usually a high level one) in the dominating cluster dominates at least one concept in the dominated cluster. Its meaning is therefore less strict than that of edges in the original lattice, and we should not mistakenly interpret it as indicating that every concept in the dominating cluster dominates every concept in the dominated cluster.

The abstraction lattice heuristics groups together related operations, even if they were separated into different concepts due to differences in their (low-level) implementation. Stated differently, we now have a *pyramid of abstractions*, where methods which use the same set of fields are in the same concept, and sets of methods which are used together are in the same cluster.

The idea of clustering concepts<sup>7</sup> is not new. Kuipers and Moonen [35] propose a different clustering technique, *refinement*, which, unlike ours, relies on the *user* to point out related concepts which are then merged by an interactive system.

To see that automatic clustering works for lattices of classes, consider concepts  $C_{22}$  (`bind atoms`) and  $C_{20}$  (`add bond`). The automatic clustering of these concepts highlights the similarity

<sup>6</sup>We add the bottom concept of the entire lattice ( $C_1$ ) to component  $L$  in order to form the complete lattice required by the abstraction context.

<sup>7</sup>Not to be confused with *cluster analysis* (e.g. [57]).

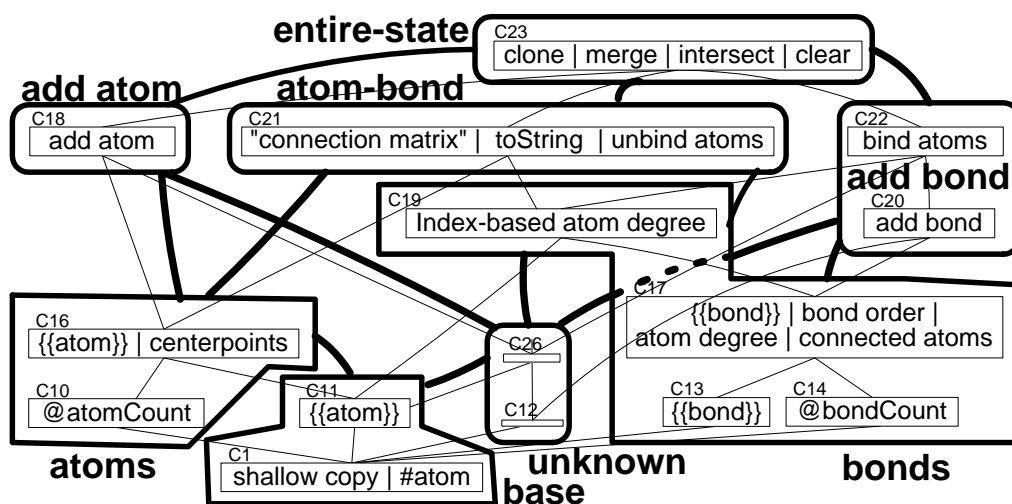


Figure 5.9: Abstraction lattice of component  $L$  of `Molecule` superimposed on the original lattice.

between the services they provide. In examining the lattice structure, it is even easy to surmise that  $C_{22}$  inserts a bond between atoms existing in the current molecule, while  $C_{20}$  may lead to an inconsistency by binding together atom objects which reside in other molecules. Another example is the cluster named `bonds`, which reveals that the notions of “atom degree” and “bond” are tightly related.

We also observe that both the `add bond` and `add atom` clusters use the two *unknown* concepts (the concepts with no **public** methods or fields), which are even clustered together. An educated guess (which is readily confirmed in stages II and III) is that *unknown* contains utilities related to resizing the two collections.

Because of the heuristic nature of the abstraction lattice, not every bit of information is significant, and can sometimes actually be misleading. For example, concepts which deal solely with atoms are scattered across the `atoms` and `base` clusters. Remember that we must also be careful in interpreting edges in the abstraction lattice. For example, although the `bonds` cluster dominates the *unknown* and (indirectly) `base` clusters, the lower concepts in `bonds` are not concerned with the “unknown” operations or with the atoms-related operations that the dominated clusters contain.

## 5.10 Step 8: Match services against expectations

It is now finally time to examine in detail the services supplied by the class, matching these against the expectations built in the first two steps. Our examination will use the pyramid of abstractions as a road map to the journey and as a directory of services.

We begin with a work list containing the expected mandatory functionality, followed by the other non-mandatory features we surmised in the earlier steps. In searching for a functionality we may begin by marking all related clusters, and proceed by zooming-in to concepts and methods, examining first their name, then their full signature and accompanying documentation (e.g., `JAVADOC` comments). Partial matches against our expectations and other discoveries made along the way are dynamically added to our work list.

For example, in searching for the *bond management* functionality, which we expected to be provided by any class that represents molecules, we mark clusters `bonds` and `add_bonds`. The `atom-bond` cluster is ignored for now because it does not deal solely with bonds. We examine the outline lattice and find out that all the concepts in the two marked clusters seem directly related to bond management. We verify this by examining the actual methods in these concepts (Fig. 5.10), while adding unrelated methods (if any) to the work list.

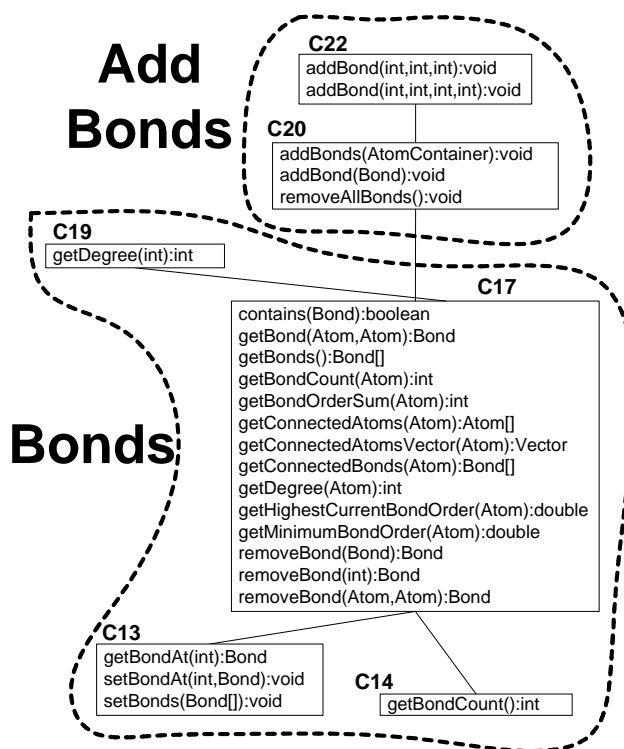


Figure 5.10: Concepts in the `bonds` and `add_bonds` clusters of `Molecule`

The pyramidal search for functionalities and the careful examination of signatures highlights inconsistencies and other design flaws. For example, in concept  $C_{17}$  of Fig. 5.10, we find that `getMinimumBondOrder` returns a **double**, while another method in the same concept, `getBondOrderSum`, which presumably computes the sum of bond orders, returns an **int**. Examining methods `getHighestCurrentBondOrder` and `getMinimumBondOrder` suggests that the class designers did not apply a consistent naming scheme. Also in the same concept we find three methods: `getDegree(Atom)`, `getBondCount(Atom)` and `getBondOrderSum(Atom)`, whose distinct responsibilities are not clear from their names nor from their accompanying documentation. We write these down for further examination.

In addition, we write down potential implementation problems, and will examine them in the next stages. For example, the structure of the `bonds` cluster suggests that concept  $C_{13}$  introduces a field that maintains a collection of bonds, and that concept  $C_{14}$  introduces one that tracks the size of the collection. If this hypothesis is correct, then the appearance of `setBonds` in  $C_{13}$  leads to an error because the field tracking the number of bonds is not updated. When we examine the fields in the second stage of the analysis of the class we will verify if this is indeed the case.

## 5.11 Step 9: Identify core, auxiliary and wrapper services

Many class designers like to follow Meyer’s “shopping list approach” [42, pp.80–83], according to which classes provide a coherent and exhaustive set of services. In trying to understand the functionality of the class, it makes sense to distinguish between *core*- and *auxiliary*- services. Roughly speaking, a service is *core* if it is essential to the usability of the class and cannot be emulated using any other services which that class provides. Auxiliary services are then thrown-in for completeness and client convenience.

Some of the auxiliary services merely *wrap* other services, adding no- or minimal- functionality in the process. *Wrappers* may, for example, negate the return value, provide a default argument, adapt the type of arguments or return value, etc.

In many cases, a wrapper will be in the same concept as the operation it wraps. Method metrics [12] such as code length or McCabe’s cyclomatic complexity [40] also provide valuable clues for the almost automatic identification of wrappers. A very typical characteristic is that the wrapper calls just one function, often with the same (overloaded) name.

It is more difficult to (semi-) automatically distinguish between core and non-wrapper auxiliary methods. One clue is that core methods tend to be in lower level concepts, make few or even zero calls to methods in the same class, and that these called methods are in very low layers.

In the next chapter, we show how peeking into the implementation gives us more clues for making this distinction, and introduce the *embedded call graph*, which is a special call-graph layout that can help a reader determine the nature of such methods.

# Chapter 6

## Methodology Stage II: Implementation Analysis

### 6.1 Introduction

When we studied the interface of a class in the previous chapter, we ignored implementation details (and occasionally errors), even if they were exposed in the interface itself. In addition, we were left with questions regarding parts of the interface which couldn't be resolved without examining the implementation. In particular, we had the *unknown* concepts which contain no interface members but are apparently significant for understanding the methods in dominating concepts. We also encountered groups of methods whose distinct purpose could not be precisely inferred from their names or documentation.

In this chapter, we begin to zoom-in into implementation details, at this stage *without* inspecting the source code. For example, we include method signatures and study the names of fields which were omitted in the previous stage. We also examine the *method call graph* which can be computed from the compiled representation. By studying the fields that each method uses and the methods that use each field, we hope to understand how the state of the class is realized, and how this state can be viewed and modified by the class methods.

One could argue that since we are already examining the implementation of the class in this stage, then there is no reason to postpone reading the source code until the next one. However, it is our belief that a reader inspecting the source code is likely to be distracted by the details and errors in the implementations of individual methods and not be able to understand the “bigger picture”, the overall structure and interface of the class. We therefore argue that before a full code inspection takes place, the overall structure should be understood and scrutinized for problems in the interfaces between methods.

There are 10 steps in this stage:

1. Construct the embedded call graph.
2. Identify unused fields.
3. Discover the roles of fields.
4. Investigate interdependencies between fields.
5. Assess the quality of the names of fields.

6. Investigate methods which access the entire state.
7. Investigate methods which do not access any part of the state.
8. Study asymmetries.
9. Study the access patterns of methods.
10. Examine non-public methods.

Some of these steps can be thought of as check-list items of code inspection. Again, and as customary in methodologies, the precedence relation between steps is not very strict.

## 6.2 Step 1: Construct the embedded call graph

Call graphs are a powerful tool for understanding the interrelationship between methods, and are integrated into many software analysis and development environments. Further, as demonstrated by the seminal work of Lanza and Ducasse [37] on class blueprints, the shape of graphs that model calls in the class can provide clues on its semantical organization.

In a similar manner, we propose the use of an *embedded call graph* (ECG), which is a novel diagram obtained by superimposing the method call graph of a class on the concept lattice of the same class, so that the node of each method is embedded in an enclosing node that represents the concept which introduces it. To illustrate this definition, consider Fig. 6.1 which depicts the ECG of class `Pnt3D`, whose concept lattice was constructed in Chapter 3.

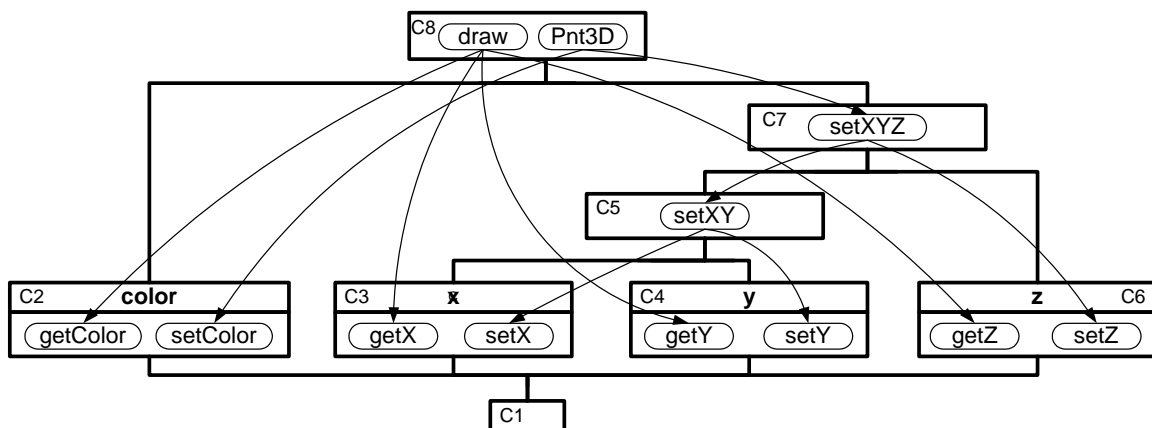


Figure 6.1: Embedded call graph of class `Pnt3D`

The ECG can be thought of as a semantic-driven heuristic for laying out the call graph since, by definition, methods can only invoke themselves or other methods which appear in the same concept or in dominated ones. Recursive and mutually recursive calls, which result in cycles in the call graph, are always limited to a single concept. Also, if the lattice is horizontally decomposable, then the edges of different components never cross.

Whereas a regular concept lattice does not imply an order between items in the same concept, the ECG creates a partial order between them, showing dependencies and hinting at levels of abstraction. This fact will be of much use in the third stage of our methodology, since it will assist us in determining a topological order for inspecting the methods of a concept. For example, consider Fig. 6.2 which

depicts an ECG limited to component  $L$  of class `Molecule`. The ECG clearly shows that `clone` invokes `removeAllElements`, and therefore the code of the former should be read only after that of the latter.

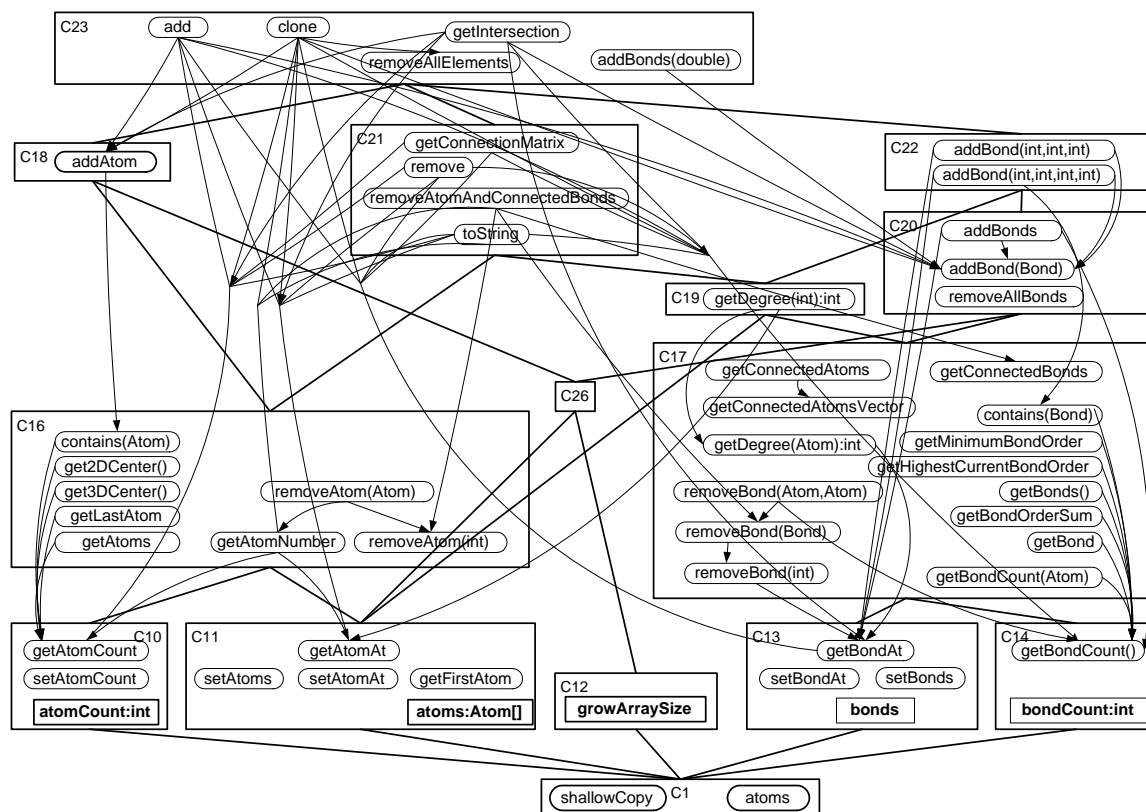


Figure 6.2: Embedded call graph of component  $L$  in class `Molecule`

In addition, we can use the ECG to determine which overloads of methods are likely to be wrappers. For example, `getDegree(int)` in  $C_{19}$  of Fig. 6.2 apparently wraps `getDegree(Atom)` in  $C_{17}$ . The two overloaded versions of `addBond` in  $C_{22}$  wrap the version in  $C_{20}$ , and there is a chain of wrapping between the three `removeBond` methods in  $C_{17}$ .

Studying the ECG also helps us surmise how some of these methods are implemented. For example, `getDegree(int)` in  $C_{19}$  apparently retrieves an atom corresponding to the given integer parameter from the array, by invoking `getAtomAt(int)` in  $C_{11}$ , and then retrieves the degree of the specified index by invoking `getDegree(Atom)` in  $C_{17}$ .

Note that in plotting the embedded call graph of a context from which some methods (e.g. **private**) are omitted, edges should indicate indirect invocations that pass through the omitted methods.

In addition to the ECG of the entire lattice, we define a *restricted ECG* of a certain concept to be the subgraph induced by the nodes representing methods of that concept. Hence, nodes representing methods in other concepts and edges leading to or from them are removed, reducing clutter. The restricted graph will be used when we zoom-in on a particular concept or when we attempt to determine an order for reading its methods.

## 6.3 Step 2: Identify unused fields

In the early stages of the development of a class, unused **private** fields are common because many methods are still stubs. Unused fields disappear as the class nears completion and the stubs are implemented. In mature classes, maintenance can result in *dead fields*, which remain in the class while the methods that use them are removed, lose their code, or become deprecated<sup>1</sup>.

If every field in a class is used by at least one method, then the top concept of the lattice contains the methods which use all the fields. If, on the other hand, there is at least one field in the class which is not used by any method of that class, then the lattice will have a top-string, and the unused fields will appear in its top concept without any accompanying methods.

In the `Molecule` class, all the **private** fields appear in the bottom layer. The lattice does have, however, a **public** field named **pointers** which is not used by any method and which appears in the top concept,  $C_{25}$ . The lack of an accessor and a mutator for this field is acceptable since the field is exposed to external clients. Nevertheless, the fact that it is not accessed by the constructor or the cloning operator (which appear lower in the lattice) suggests that this field is either “dead”, or that there is an error in the implementation of these methods.

## 6.4 Step 3: Discover the roles of fields

In order to understand the implementation of a class, we must first understand the structure of its state as defined by its fields. Most of these fields are non-**public** and are now restored, after having been omitted from the first stage where we concentrated on the interface.

The fields introduced in the trivial components of the lattice are the easiest to study because they are independent from one another. The methods which accompany a field are usually enough to infer its role or to verify that its name is consistent with its role. For example, concept  $C_3$  in Fig. 6.3 introduces a vector field named `chemObjects`. Only the accompanying methods, `addListener` and `removeListener`, tell us that this field actually stores a collection of `ChemObjectListener` objects and not of `ChemObject` objects. Further examination of Fig. 6.3 shows that the roles and names of the other independent fields in `Molecule` are clear and consistent.

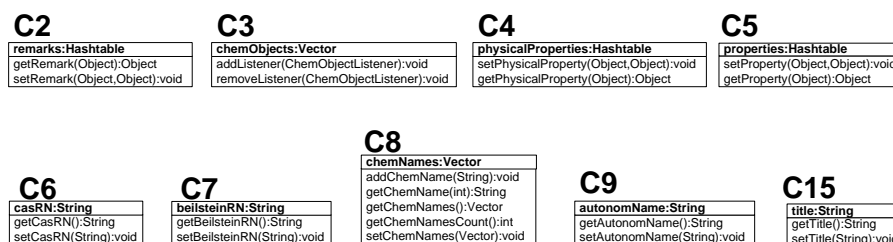


Figure 6.3: Trivial components in the lattice of `Molecule`

Things are more complicated with fields introduced in nontrivial components because their roles are likely to be carried out in conjunction with other fields. For example, consider Fig. 6.4 which depicts component  $L$  of `Molecule` and focus on the five fields introduced in its bottom layer. The

<sup>1</sup>Note that it is possible to use a context which removes JAVA methods and fields marked as **deprecated**.

meaning of the two arrays, `atoms : Atom[ ]` and `bonds : Bond[ ]` is obvious; they are the collections which respectively store the atoms and bonds of the molecule. The names of the two “count” fields, `atomCount : int` and `bondCount : int` suggest that they track the number of items in the collections. Note that these fields are different from the `length` properties of the arrays, which track the total allocated size.

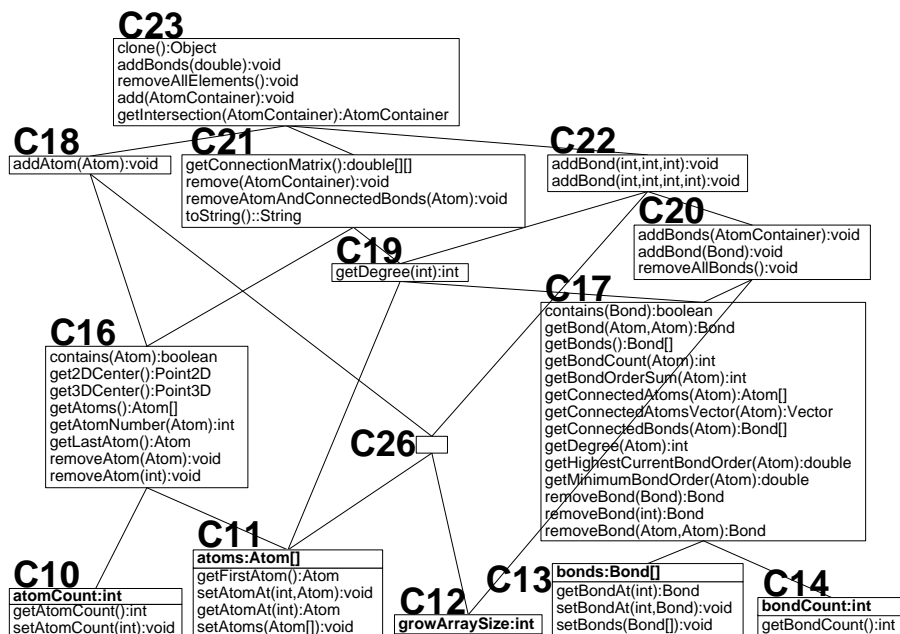


Figure 6.4: A zoomed-in concept lattice of component *L* of *Molecule* with fields.

The role of `growArraySize : int`, introduced in *C*<sub>12</sub>, is even more difficult to deduce because there is no method which uses solely that field. We examine the directly dominating concepts and discover that they all deal with the addition and removal of atoms and bonds. Based on this field’s name, we can guess that *Molecule* dynamically grows the arrays of bonds and atoms, and that this field specifies the chunk size. Surprisingly, in redrawing the lattice based on a write-access context, we find that `growArraySize` is *modified* when inserting atoms and bonds. We make a mental note to check during code inspection why the chunk-size should change.

## 6.5 Step 4: Investigate interdependencies between fields

The layer-structure of a lattice, and in particular its non-empty second-layer concepts, highlights *strong ties between fields*. We already saw that these ties are important for understanding the roles of the fields, but they can also help us discover the invariants of the class.

In the lattice of *Molecule*, the only non-empty second-layer concepts are *C*<sub>16</sub> and *C*<sub>17</sub> in component *L* (Fig. 6.4). These concepts reveal that the `atoms` and `atomCount` fields are very closely related, and that so are `bonds` and `bondCount`. The interdependency between these pairs of fields is also indicated by a similarity in names.

A moment’s pondering raises the suspicion that the information on the number of atoms (and bonds) is redundant because it can be calculated from the number of non-empty entries in the arrays.

In fact, together the array and count fields essentially implement a resizable array, not unlike the standard `Vector` class. According to the authors<sup>2</sup> of CDK, micro-management of a resizable collection was chosen over one of the standard collection types for performance reasons. Nevertheless, such an implementation complicates the class and introduces a new invariant, that the value of the count field must always be equal to the number of non-empty array entries. As we shall later see, the `Molecule` class fails to preserve this invariant.

In addition to the case where fields in two bottom-layer concepts are connected by a second layer concept, we mention here the special cases of *implications* between fields and of several fields introduced in the same concept.

If a field  $f_1$  is introduced in some concept, and that concept dominates another concept which introduces a second field  $f_2$ , then we say that field  $f_1$  *implies*  $f_2$  because if the former is used by a method, then the latter must also be used. Although an implication might be purely coincidental, in other cases it might indicate a strong connection between the fields. To demonstrate this, consider Fig. 6.5 which depicts several concepts from a class named `Graph`, which we shall explore in Chapter 8.

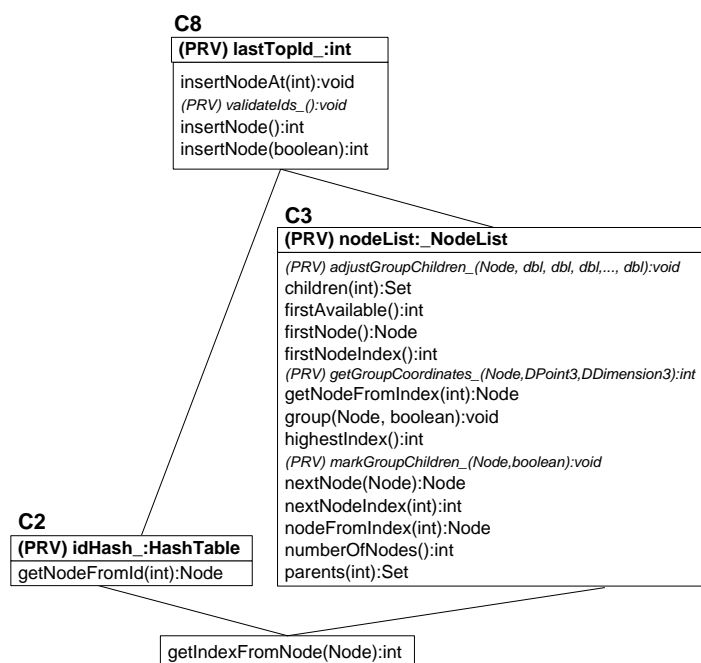


Figure 6.5: Implications between fields in class `Graph`.  
(Only relevant concepts are shown)

Concept  $C_2$  in the bottom layer of Fig. 6.5 introduces a field that collects nodes in a list, and concept  $C_3$  introduces a field that collects nodes in a hash-table. Concept  $C_8$  in the second layer connects these two concepts, introduces methods for adding new nodes (affecting both collections), and introduces a field that provides the unique identifiers for the hash. Hence, the implication between the `lastTopId` field and `idHash` follows the functional connection between these two fields, whereas the implication between `lastTopId` and `nodeList` is coincidental and occurs only because both are used whenever nodes are added to the graph.

<sup>2</sup>Dr. Christopher Steinbeck, personal communication

The strongest connection between fields is suggested when at least two fields are introduced in the same concept. Since the fields imply each other, they are always either used together or not used at all. However, the more details omitted from the context, the less likely the fields are to be indeed related. For example, consider a class where all fields have **private** inspectors and mutators. In the lattice of the complete context of such a class, every field will be introduced in a different concept, and the connections between fields will be evident only in the second layer. In a lattice of a context that omits **private** members, the bottom layer will be removed and the fields will be introduced in the concepts of the original second layer, without implying any stronger connection. This suggests that if we see too many fields introduced in the same concept of a lattice for a detailed context, then accessors for individual fields might be missing and should be added.

## 6.6 Step 5: Assess the quality of the names of fields

When the role of each field is more or less understood, we should check that fields are named properly. In particular, we should verify that the names of fields indicate their purpose (e.g. “mappingFromX-toY”) rather than a specific implementation detail which can change over time (e.g. “hashTableOfY”). For instance, in the lattice of `Molecule` we saw concept  $C_3$  which maintains “listeners”, operates on a vector field named `chemObjects`, and inserts objects of type `ChemObjectListener`. The name of this field should probably be changed to `chemObjectListeners` to indicate that it is a simple collection of special-purpose listeners. Note that the array-like nature of the collection is not manifested in the choice of name because the accompanying methods indicate that listeners are accessed by value and not by index.

## 6.7 Step 6: Investigate methods which access the entire state

Some methods, such as constructors, cloners, and serializers, are intended to operate on the entire state of the object. These methods, which we shall call *entire-state methods*, are therefore expected to appear in the top concept of the lattice. Exceptions to this rule (where such a method appears in a lower concept) should be closely investigated.

If all the concepts which dominate the *entire-state* method’s concept do not introduce additional methods, then the fields introduced by these concepts are apparently “dead”, and the method actually uses all the effective fields. However, if the dominating concepts do introduce methods (and possibly fields) then the method clearly misses some of the effective fields. It is acceptable to miss fields if they are redundant or used only for transient purposes, such as performance enhancement. For example, if the class includes a field that serves as a cache of recently used collection entries, then that cache is likely not to be used in cloning or comparisons. In other cases, missing a field suggests an error in the implementation of the method.

We already saw that the constructors of `Molecule` appear in concept  $C_{24}$  (see Fig. 5.6) which does not dominate the `pointers` and `title` fields. We now turn to the other entire-state methods to search for similar errors. The `Molecule` class has no `copy` method, but it provides a `shallowCopy` method in the bottom concept. This method accepts no parameters and returns a value of type `Object`, suggesting that this is actually a `shallowClone` method; the accompanying `JAVA-`



**static methods** Methods declared as **static** cannot use any non-**static** fields, but often do not use any **static** fields either. Such methods are very common in JAVA, because the language does not support macros or non-member functions and forces the use of **static** methods for these purposes.

**abstract methods** If **abstract** methods are not removed from the context, then they will naturally appear in the bottom concept of the lattice.

**inheritance hooks** Programmers sometimes leave hooks for optional implementations in inheriting classes by defining a non-**abstract** method with an empty body or stub code.

**constant returners** Occasionally, programmers need to provide different values for a particular constant in different classes of a hierarchy. For example, in a class hierarchy representing file types, a different default filename extension might be associated with each class. The usual JAVA idiom for realizing constants in JAVA (i.e. **final static** variables) cannot be used in such cases because a static variable declared in a superclass is shared in all the subclasses. Instead, a non-**static** method which simply returns a different constant value is implemented in each class. We refer to such methods as *constant returners*; naturally, they use no fields.

**methods which indirectly refer to fields** Due to the limitations of the static analysis we perform,<sup>4</sup> it is possible for a method to indirectly refer to a field, perhaps via an alias or another class, so that the access pattern of the method would not be recorded correctly. The code of the method must be examined to determine if this is indeed the case.

**deferred access to fields** Consider, for example, the following implementation of an iterator (or enumerator) for a collection field: When the iterator is first created, it simply stores a reference to the creating object. The collection field is only accessed if and when the `next` method is invoked on the iterator. We say that the access to the field is *deferred*, and therefore not counted in calculating the fields accessed by the method.

**native methods** Methods defined as **native** are not analyzed by our static analyzer but could, potentially, affect fields. For example, the `clone` method of `Object` is declared as **native**, but creates a new object so it affects all the fields of the new object. Nevertheless, our calculation of the accesses relationship would not detect this use of fields.

Both of the bottom-concept methods of the `Molecule` class are apparently there for legitimate reasons. We already speculated that the first, `shallowCopy`, invokes the **native** implementation of `clone` from `Object` and is therefore erroneously listed as not using fields. The other method, `atoms`, returns an enumeration of atoms, and probably accesses the fields when the `nextElement` method is invoked.

In other cases, however, the fact that a method uses no fields indicates a real problem. For example:

**stubs and dead-code** Early in the development of a class, many methods are stubs; they do not contain meaningful code or use any fields. In a mature class, such methods might be *dead methods* which were not removed due to neglect or due to the need to preserve an existing interface. Such methods should be marked as **deprecated** and modified to return an exception.

---

<sup>4</sup>The literature on static analysis discusses ways to deal with indirect references, but this discussion falls outside the scope of this work.

**undeclared statics** In many cases, a method which should be declared as **static** is wrongfully not declared so.

**problematic refinement** On some occasions, programmers who wish to refine an inherited method neglect to make the actual refinement call (using the **super** keyword), or more commonly, write only the refinement call. The latter may happen if at the time of writing the overriding method, the subclass did not yet declare new fields.

## 6.9 Step 8: Study asymmetries

As we noticed earlier with the concept lattice of class `Print3D`, asymmetries in the class lattice can be very telling; they can help discover problems such as incomplete interfaces, inappropriate use of inheritance, etc.

In the `Molecule` class, we expect to see a lot of symmetries between the methods for managing atoms and bonds. In particular, we expect to see symmetries between the methods dealing with the array fields (concepts  $C_{11}$  and  $C_{13}$  in Table 6.1), between those dealing with the count fields ( $C_{10}$  and  $C_{14}$ ), and between those dealing with both ( $C_{16}$  and  $C_{17}$ ). We also expect to see both symmetries and asymmetries between the methods dealing with addition ( $C_{18}$ ,  $C_{20}$  and  $C_{22}$ ).

	Arrays	Counts	Arrays And Counts	Additions
Methods for Atoms	<b>C11</b> <code>atoms:Atom[]</code> <code>getFirstAtom():Atom</code> <code>setAtomAt(int,Atom):void</code> <code>getAtomAt(int):Atom</code> <code>setAtoms(Atom[]):void</code>	<b>C10</b> <code>atomCount:int</code> <code>getAtomCount():int</code> <code>setAtomCount(int):void</code>	<b>C16</b> <code>contains(Atom):boolean</code> <code>get2DCenter():Point2D</code> <code>get3DCenter():Point3D</code> <code>getAtoms():Atom[]</code> <code>getAtomNumber(Atom):int</code> <code>getLastAtom():Atom</code> <code>removeAtom(Atom):void</code> <code>removeAtom(int):void</code>	<b>C18</b> <code>addAtom(Atom):void</code>
Methods for Bonds	<b>C13</b> <code>bonds:Bond[]</code> <code>getBondAt(int):Bond</code> <code>setBondAt(int,Bond):void</code> <code>setBonds(Bond[]):void</code>	<b>C14</b> <code>bondCount:int</code> <code>getBondCount():int</code>	<b>C17</b> <code>contains(Bond):boolean</code> <code>getBond(Atom,Atom):Bond</code> <code>getBonds():Bond[]</code> <code>getBondCount(Atom):int</code> <code>getBondOrderSum(Atom):int</code> <code>getConnectedAtoms(Atom):Atom[]</code> <code>getConnectedAtomsVector(Atom):Vector</code> <code>getConnectedBonds(Atom):Bond[]</code> <code>getDegree(Atom):int</code> <code>getHighestCurrentBondOrder(Atom):double</code> <code>getMinimumBondOrder(Atom):double</code> <code>removeBond(Bond):Bond</code> <code>removeBond(int):Bond</code> <code>removeBond(Atom,Atom):Bond</code>	<b>C22</b> <code>addBond(int,int,int):void</code> <code>addBond(int,int,int,int):void</code> <b>C20</b> <code>addBonds(AtomContainer):void</code> <code>addBond(Bond):void</code> <code>removeAllBonds():void</code>

Table 6.1: Symmetric concepts in the lattice of the `Molecule` class.

First, we compare  $C_{11}$  and  $C_{13}$  which respectively introduce the `atoms` and `bonds` fields. We see that the `getFirstAtom` method in  $C_{11}$  does not have a corresponding `getFirstBond` method in  $C_{13}$ . Even though this method is likely to be only a wrapper, its existence for one field and absence for the other can confuse clients.

Next, we compare  $C_{10}$  and  $C_{14}$  which respectively introduce `atomCount` and `bondCount`. We immediately notice that the first introduces a mutator and an inspector, whereas the other provides only an inspector. It seems that the inclusion of the mutator as a **public** method is a mistake, because its use can break the invariant that the number of non-empty entries in the `atoms` array must always be equal to `atomCount`. We surmise that the mutator doesn't preserve this invariant because it does not affect the `atoms` field, as obvious from its location in the lattice.

We now turn to  $C_{16}$  and  $C_{17}$ , which combine the array and count fields. We expect to see here many methods specific to either atoms or bonds, but also methods that should be symmetric in both. Indeed, both concepts contain methods for checking array membership, retrieving the entire array, and removing elements from it. However, concept  $C_{16}$  contains a `getLastAtom` method for which no corresponding `getLastBond` method appears in  $C_{17}$ .<sup>5</sup> In addition, if we zoom in and include details about thrown exceptions, we learn that some of the `remove` methods for atoms declare a `NoSuchAtomException` exception whereas the corresponding removal methods for bonds do not declare a symmetric exception.

Finally, we examine the difference between the `addAtom` method and the various `addBond` methods. Atom addition is apparently straightforward: the array is enlarged (if necessary) and the `Atom` object is added to the array. Bond addition, on the other hand, comes in two flavors. An existing `Bond` can be added in a manner similar to that for atoms, or a new bond can be created between existing atoms, specified by their indices. This retrieval causes the access to the `atoms` array, which results in the separation into two concepts.

Note that some of the symmetries mentioned above were already evident when we studied the interface in the first stage, but were ignored since we were not able to confirm that certain concepts should be symmetric before seeing the fields.

## 6.10 Step 9: Study the access patterns of methods

The next-to-last step involves, based on the information collected so far, meticulously checking that every method uses precisely the fields that it should. By eliminating read- or write- entries from the context we can also verify that the method makes the expected kind of access to each field.

We already noticed flaws in the access patterns of the special methods of `Molecule`, but such problems exist in some of its other methods. For example, the invariant that the value of `atomCount` should always be equal to the number of non-empty entries in `atoms` is apparently broken by `setAtoms` in  $C_{11}$ : According to its name and documentation, this method replaces the previous array of atoms with a new one. Because it appears in  $C_{11}$ , this method seems to rewrite `atoms`, without updating the value of `atomCount`. A symmetric problem occurs with `setBonds` in  $C_{13}$ . The same invariant may also be broken if `setAtomAt` in  $C_{11}$  is used on a slot which was previously empty, or if `setAtomCount` ( $C_{10}$ ) is used by a client. The exposure of this method to clients is therefore a clear breach of both the encapsulation and the safety of the class.

Another problem which we discover by examining the access patterns of methods is that the removal of an atom (in  $C_{16}$ ) does not cause the removal of the incident bonds. After an atom is removed, the molecule may still contain a bond that binds the removed atom.

## 6.11 Step 10: Examine non-public methods

To fully understand a class, we must consider methods which do not take part in the interface, and therefore examine a lattice based on a context in which all non-**public** methods are present. Fig. 6.7

---

<sup>5</sup>In fact, the method in  $C_{16}$  should probably be removed because there should be no ordering between atoms from the point of view of the client.



- The `setAtoms` method in  $C_{11}$ , which replaces the entire set of atoms, fails to update the `atomCount` field in  $C_{10}$ . External use of this method may lead to the breaking of the class invariant, that the value of `atomCount` must always be equal to the number of valid entries in the `atoms` array. The `setBonds` method in  $C_{13}$  causes the same problem for bonds.
- The `setAtomAt` method in  $C_{10}$  does not affect the `atomCount` field, potentially breaking the class invariant. A similar problem occurs with `setBondAt` in  $C_{13}$ .
- The `setAtomCount` method in  $C_{10}$  is exposed as a **public** method. External use of this method may break the class invariant.
- If a new bond object is added to the molecule using the `addBond(Bond)` method in  $C_{20}$ , then there is no guarantee that the bond actually connects atoms which are members of the molecule.
- The removal of an atom from a molecule using the `removeAtom` methods in  $C_{16}$  does not remove the connected bonds. As a result, some bonds may refer to atoms that are no longer in the molecule. Note that in the symmetrical case of the removal of a bond, there is no need to remove the connected atoms because a molecule object is incrementally built by adding atoms and then binding them with bonds.

## Interface problems

- The implementation of the collections of atoms and bonds as two arrays is exposed in the interface. In addition, the `flags`, `pointers`, and `title` fields are exposed as **public** members.
- Some of the methods dealing with *bond order* in  $C_{17}$ , such as `getMinimumBondOrder` and `getHighestCurrentBondOrder` return a **double** value for the order, whereas others, such as `getBondOrderSum`, return an **int**.
- There is a naming inconsistency in  $C_{17}$  between `getMinimumBondOrder` and the apparently symmetrical `getHighestCurrentBondOrder`.
- The `title` field in  $C_{15}$  is exposed as a **public** field even though it has an inspector and a mutator.
- The part of the interface dealing with chemical names ( $C_8$ ) does not support the removal of a single name.
- There are asymmetries between the part of the interface dealing with atoms and the part dealing with bond. For example, the methods `getFirstAtom` and `getLastBond` have no bond counterparts. Also, there is a `setAtomCount` (that should be removed) but no `setBondCount`, and `removeAllBonds` but no `removeAllAtoms` (which is realized by the `removeAllElements` method).
- The `shallowCopy` method in  $C_1$  should be named `shallowClone`! It accepts no parameters and returns an `Object`, like a cloning operation, instead of accepting another `Molecule` and not returning anything, like a copy operation. Also, there are `shallowCopy` and `clone` operations, but no `copy` and `shallowClone`.
- The distinction between three methods in  $C_{17}$ : `getBondCount(Atom)`, `getBondOrderSum(Atom)` and `getDegree(Atom)` is not clear, and it is possible that some of these methods are redundant. Their code must be inspected to determine if this is indeed the case.

- The distinction between `getBondCount()` and `getBondCount(Atom)` is not clear.
- The existence of `getConnectedAtomsVector()` ( $C_{17}$ ) in addition to the method `getConnectedAtoms()`, which returns an array, seems superfluous.

### Style problems

- The name of the `chemObjects` field ( $C_3$ ) is inappropriate, because it actually stores a collection of `ChemObjectListener` objects.
- The `growArraySize` field is shared for both the array of atoms and the array of bonds, although each may grow at a different pace.

In order to fully understand the class, to verify some of the detected problems, and to discover additional ones, the next chapter will describe the third stage, where we inspect the actual source code of the class.

# Chapter 7

## Methodology Stage III: Code Inspection

### 7.1 Introduction

The simple technique of reading source code is used extensively for software understanding and maintenance. Reading source code is at times the only way to understand an undocumented system. It is also a common means of verifying the correctness and quality of code, an activity known as *code inspection* [19, 24].

In the previous chapters we demonstrated that concept lattices can be of assistance in studying the interface and implementation of a class. We now show that these lattices can also be of use in inspecting the code of its methods. The lattice helps us select an order of reading the methods which improves the quality of the results and decreases reading time.

We begin by discussing the problems involved in inspecting the code of object oriented systems and individual classes, and proceed to propose a methodology consisting of a reading order and inspection tasks which should accompany it.

### 7.2 The difficulties of inspecting object oriented systems

The problem of inspecting the code of an object-oriented system is considered in the literature to be much more difficult than inspecting that of a procedural system. For instance, Dunsmore, Roper, and Wood [13] argue that inheritance, dynamic binding, and small methods all exaggerate delocalization effects, thereby splitting the source code into small units spread across many source files.

Code inspection is usually performed on entire systems, but even the inspection of a single class is far from being trivial. The size and complexity of certain classes can overwhelm a reader, allowing various defects and replications to go unnoticed. In addition, there is a problem of mutual recursion that makes reading the code of a single class more difficult than reading that of a single module. This problem has to do with the evolution from procedural languages, in which mutually recursive definitions are rare, to object oriented languages, in which such definitions become commonplace.

In most procedural languages (e.g. C, PASCAL, and ML), there are few cases in which mutually recursive definitions are required. These rare cases are resolved using *forward declarations* (as with the **forward** keyword in PASCAL), or using *co-definitions* [59] (as in ML [44]). In contrast, in the definitions of classes in object oriented languages, co-definitions are the rule rather than the exception.

This difference is the reason for one of the major changes in the evolution from C to C++. In C, the definitions within a module are sequential, and the module is processed incrementally by the compiler. The scarcity of mutual recursions allows the compiler to process the entire module in a single pass. It also facilitates reading the source code of the module, since incremental understanding is within the capabilities of the human mind. The situation is different in C++ and in all the other major object oriented languages, including JAVA and EIFFEL. The definitions of a class are of the co- rather than of the sequential kind: all class members are considered to be defined at the same time.

The change in semantics is not only due to technical reasons. It expresses a shift in the perspective of language designers, which inevitably propagates to the design and analysis of classes. Supporting the co-definition semantics of the entire class requires the compiler to manage large open symbol tables, and perhaps perform multiple passes for consistency checks. While this requirement is not too demanding for today's modern compilers, an individual who is asked to do the same is faced with the limitations of the human memory, which cannot maintain the required large multi-visit multi-update buffers at the same ease.

### 7.3 Reading order

Empirical research (e.g. [14]) confirms that the order in which the source code of object oriented systems is read can have a significant effect on the efficiency of code review and inspection, and on the quality of their results. We believe that this effect is significant even when only a single class is inspected in isolation, as we attribute the impact of the reading order to the limitations of the human mind. Consider, for example, a class with dozens of methods in which two methods have the same functionality but different names. We will certainly discover this redundancy if we read these two methods consecutively; our prospects are lower if we read many other methods in between them.

For every class, we would ideally like to be able to select a reading order that optimizes the inspection process. Unfortunately, "great minds do not necessarily think alike": The subjective nature of understanding makes the existence of a universally optimal reading order unlikely. Furthermore, defining an optimal reading order might require knowledge which can only be gained by reading the entire class in the first place.

Reading methods in the order of their appearance in the source file is not practical for large classes. As a class matures and undergoes ad-hoc changes, any predetermined order which might have existed is likely to disappear. Also, the order in which members appear in the source code of a class is not necessarily meaningful since members are assumed to be defined concurrently. Moreover, programming language mechanisms such as inlining can actually spread related methods across several source files, for example in the case of an **inline** method invoking a non-**inline** method in C++.

In planning an alternative effective order for reading a class, we hope to make related methods appear together, thereby increasing the probability of detecting duplications and opportunities for code sharing. Another objective is to reduce the mental load off the human inspector by offering a hierarchical organization and by minimizing the number of *forward references*. This objective is also served by the *simplest-first rule*, by which, if there is no particular order between several items, then they are inspected in an ascending complexity order. The rationale behind this rule is obvious: First, the reader can process a sequence of very simple items efficiently. Second, the reader has to memorize

fewer bits of information on average through the reading process. This is similar to the rationale for placing the shorter block first in an **if-then-else** clause.

We propose a lattice-based inspection order, by which the code of all the methods introduced in the same concept is read together without interleaving with methods of other concepts. We argue that methods appearing in the same concept tend to be similar in their purpose, semantics and implementation, and reading them consecutively is therefore more effective. For example, because such methods are likely to use a common set of library classes, a non-interleaved reading order minimizes the amount of information the reader has to track at the same time and reduces the chance of information being forgotten.

The idea of reading methods according to concepts is similar to Meyer’s suggestion [41] that methods be organized in groups by responsibility (*global order*), and then sorted lexicographically within each group (*local order*). The difference is that we use concepts for the responsibility-driven global order, i.e., for automatic *feature categorization*, and calling-patterns for the local order. In the remainder of this work we refer to the consecutive reading of the methods of a certain concept as *reading the concept*.

We proceed to define our proposed global and local orders.

### 7.3.1 Global order

The global order in our methodology refers to the order in which concepts are selected for reading. The following guidelines dictate this order:

#### **Read each component of a horizontally decomposable lattice separately**

In a horizontally-decomposable lattice, the concepts of each component should be read together. Since each component is independent (i.e., its methods never access fields and methods in other components), and because it (supposedly) represents a different functionality, we can “divide-and-conquer” the problem of reading the code by treating each component as a lattice in its own right.

#### **Read concepts in an ascending order of layers**

In Chapter 5 we discussed layers and noted that methods in the same layer tend to have a similar level of abstraction. We should therefore read the concepts of each layer consecutively, to benefit from the similarity between these methods. The layers themselves will be examined in a bottom-up order, because it does not conflict with the topological order between the methods. This is necessary to minimize *forward references*.

#### **Inside a layer, read concepts of the same cluster consecutively**

Our overall reading order does not process entire clusters consecutively because concepts are read by layers. Nevertheless, inside each layer we can benefit from reading the concepts of each cluster consecutively because of their similarity in purpose and infrastructure.

#### **In selecting between otherwise equivalent concepts, select the simplest first**

This guideline follows the *simplest-first* rationale of our proposed reading order. Defining the simplicity of concepts is left at the discretion of the user, but a straightforward approach is to simply select the concept which introduces the smallest number of methods. More elaborate definitions are

possible, using properties of the methods (such as the sum of the lengths of their methods) or of the interactions between them (i.e., cohesion in the ECG of the concept).

### 7.3.2 Local order

Once we have selected a concept, we should select an order for reading its methods. Consulting a *restricted ECG* limited to the methods of the selected concept is very useful in selecting this order.

#### **Read methods in a topological order**

A topological order minimizes the cases where a reader encounters an unfamiliar method, although it does not prevent them completely because of the possibility of mutual recursion. When following such an order, the reader is theoretically able to read the entire code of the method consecutively, without having to refer to the code of other methods, unless their details were already forgotten. A reverse-topological (top-down) order, on the other hand, would require the user to mentally maintain a long “stack” of calls and return locations. Whereas computers can do this efficiently, humans are prone to forgetting details and having to restart reading some methods from scratch. Nevertheless, some readers prefer a top-down order and are allowed to use it as long as they are consistent.

Note that a topological order between all the methods of the class implies a topological order between concepts, since a method can only be invoked by itself, by a method in the same concept, or by a method in a dominating concept. Hence, this guideline conforms to the global ordering between the layers. If we use a reversed-topological order here, then layers should be read in a descending order.

#### **If the restricted ECG of the concept is not connected, read each graph component separately**

It is common to see chains of invocations between methods inside a concept. For example, a wrapper often appears in the same concept as the operation that it wraps. If we restrict the call graph to methods inside the concept, we often get an unconnected graph such that the methods in each component are related. Even if the relation is only in implementation and not in functionality, reading in this order minimizes the time between reading an invoked and an invoking method.

#### **In selecting between otherwise equivalent concepts, select the simplest first**

Analogous to the guideline for selecting concepts, the strategy for selecting methods follows the same simplest-first rule. Reading methods in an ascending order of simplicity is especially important because JAVA classes sport many short and simple methods. These methods can be very efficiently read in sequence if no complex method “halts the pipeline”.

Although defining what makes methods simple and easy to understand is subjective, combinations of method metrics [12] can be used to estimate the simplicity of a method. Following are several of the metrics that can be easily obtained from a compiled JAVA classfile:

**Bytecode length (BClen)** The standard bytecode representation of JAVA serves as a single target language for JAVA compilers on all platforms. This allows the use of the *bytecode length* as a platform-independent metric for estimating the complexity of a compiled method. Unlike the commonly used *lines of code* metric, which is very susceptible to formatting differences, BClen is (relatively) immune to such differences, although not immune to compiler optimizations.

**McCabe Cyclomatic Complexity (MCC)** The number of possible execution paths through a method is manifested in the MCC [40] metric. Branching significantly increases the effort required to understand a method since the reader must understand each individual execution path. For the purpose of selecting a reading order, the calculation of MCC for a method should not be recursive. In other words, the method flow graph should not contain subgraphs for invoked methods. For example, a method that simply calls another method should be *linear* and have an MCC value of 1. Linear methods are the simplest to understand and in fact constitute the majority of methods in many JAVA applications [23]. Moreover, since they tend to follow a limited set of *nano-patterns*, methods in a lattice can be tagged with their associated-patterns, reducing the need for reading the actual source code.

**Number of method invocations (NMsg)** A method which invokes multiple methods is harder to understand than a “self-contained” method, because the reader must recall the semantics of every invoked method. The problem is aggravated when ensuring program correctness, because the pre- and post- conditions imposed by the invoked methods must also be recalled.

Empirical research [12] shows that the majority of JAVA methods are linear and very short. This fact is attributed not only to the common programming practice of using short methods in object oriented languages, but also to properties of the JAVA language itself. First, JAVA does not allow default arguments to methods, whereas C++ provides such a feature. As an alternative, programmers overload the required method name with additional methods that invoke the original method and supply these default values. Second, JAVA does not provide macro capabilities (which, although deprecated, are commonly used by C++ programmers). Instead, JAVA methods are used to implement “macro functionality” such as creating aliases.

## 7.4 Inspection tasks

Applying the many steps of stages I and II to any individual class generates many questions which can only be answered by code inspection. For example, we deferred to code inspection the resolution of issues such as the exact role of the `growArraySize` field, the differences between similar methods in concept  $C_{17}$ , and the assurance that the class invariants are held. These are in fact *class-specific inspection tasks*, coming on top of the ordinary general purpose tasks and rules of the inspection strategy.

To all these, we add a number of *general* inspection tasks, inspired by the concept lattice approach.

### Try to find duplicate services inside each concept

The rationale behind incorporating this task is that if two methods provide the same functionality, they would use the same set of fields even if their implementations are different, and therefore appear in the same concept. In our running example, we find that  $C_{17}$  in `Molecule` (see Fig. 6.4) introduces methods `getDegree(Atom)` and `getBondCount(Atom)` which despite the name dissimilarity supply exactly the same service.

### Try to identify code-sharing opportunities inside each concept

By a similar rationale, if two methods carry out a similar computation that has a potential of code sharing, then they will appear in the same concept or in neighboring ones. In  $C_{22}$ , `addBond(int, int, int)` and `addBond(int, int, int, int)`, although serving slightly different purposes, have almost identical implementation that suggest code sharing.

### **Verify that the methods in the lower concepts are not bypassed by higher ones**

Because the layers of concepts roughly correspond to the levels of abstraction of their methods, and since every method can only invoke methods in the same concept or below, we can often identify *bypassed low-level methods* by tracing ascending paths of concepts in the lattice.

OOP evangelists advocate that methods should be used even for simple operations such as field access or delegation. Naturally, such methods tend to appear in low-level concepts and are expected to be used by those in higher concepts. However, neglect and even misguided temptation to optimize, are probably the reasons for the many cases where higher level methods *bypass* the lower methods.

In this inspection task we search, especially in higher concepts, for code fragments which could be replaced by (existing) low-level methods. We can manually feed the patterns of methods from low-level concepts into a *star diagrams engine* [33], but a more automated approach is to examine the edges emanating from a high-level concept in the ECG. If the number of these edges is small, then it is likely that methods are indeed bypassed.

For example, consider again the ECG of `Molecule` in Fig. 6.2. Every method in  $C_{16}$  and  $C_{17}$  uses two fields, and we expect it to use the appropriate accessors and mutators in the bottom layer. However, as we can see in the figure, most of them use the methods for one field and access the other field directly.

# Chapter 8

## Case Study: Applying the methodology to the Graph class of VGJ

### 8.1 Introduction

In this chapter we demonstrate the use of our methodology on another real life example, a class for representing graphs from an applet called VGJ (Visualizing Graphs with JAVA). Because the methodology has already been described, the presentation here will be less formal and will demonstrate how the methodology is actually applied.

VGJ [2] is a popular JAVA applet for drawing graphs, testing layout algorithms, and animating graph algorithms. The program was originally developed at the graph drawing research group at Auburn university. It was distributed with source code under a GNU [25] public license and gave rise to numerous adaptations and extensions at different institutions. With the advent of FLGL, the commercialized version of VGJ, the original tool was discontinued although adapted versions are still in use.

VGJ is comprised of four primary packages:

**algorithm** Provides an interface for the development of graph algorithms, and several implemented examples.

**graph** Consists of classes for representing graphs and graph elements.

**gui** Contains classes for maintaining a graphical user interface using the standard AWT library.

**util** Includes several utility classes for representing elements such as points and matrices.

We choose to focus on the `graph` package because its classes are used in the interactions between the `gui` and `algorithm` packages. We will investigate the `Graph` class, which is arguably the most important class in its package (and perhaps in the entire program), since every developer who wishes to make modifications to VGJ or to add new layout algorithms must deal with this class. The `Graph` class is also interesting since it is complex and sports a very wide interface, and because graphs are a straightforward notion that recurs with different implementations in various programs.

Our focus in this case study will be on studying the interface of the class in order to discover undocumented restrictions and to surmise how the graph is represented internally. We will then examine the actual implementation to determine what class invariants must be kept and to discover

some of the optimizations that take place in the class. The analysis of the `Graph` class is going to roughly follow the steps described in Chapters 5 to 7.

## 8.2 Stage I - Analyzing the interface

### 8.2.1 Preliminaries

Just like the `Molecule` class, `Graph` represents an entity which should be familiar to most readers, a mathematical graph consisting of *nodes* (or *vertices*) and *edges* (or *connectors*). The primary functionality of this class would therefore be to manage collections of nodes and edges, much like the management of atoms and bonds in `Molecule`. We do not know, however, whether the graph is directed and whether it is simple (i.e., without parallel edges). The class documentation does not provide answers to these questions, so we will have to infer this information from studying the interface.

We delineate the responsibility of the class and do not expect it to represent special families of graphs, such as trees, although this functionality might be supplied by subclasses. In addition, although a graph can be empty or consist of a single node or a pair of nodes connected by a single edge, we expect to see other classes for representing nodes and edges.

Our familiarity with graph theory provides us with a verbose vocabulary and many possibilities for additional features, such as calculating the *in-* and *out-degrees* of edges, checking for graph *planarity*, and supporting *search* and *traversal*. If the graph is directed, we might also have method for reversing its edges or for making them undirected. Similarly, a non-simple graph might provide methods for removing parallel edges. In conjunction with the classes that represent nodes and edges, `Graph` may also provide facilities for annotating or assigning weights to the graph elements.

We now turn to exploring the environment of the class. The `Graph` class does not implement any interface, nor does it inherit from any class (except `Object`). It does, however, have a subclass named `BiconnectGraph`, confirming that special families of graphs are implemented by inheritance.

As expected, the `graph` package contains classes named `Node` and `Edge` for representing the graph elements. Two other classes in the package, `NodePropertiesDialog` and `EdgePropertiesDialog`, might hint that graph elements can be marked or annotated. A class named `GMLlexer` suggests that `VGJ` provides some textual file-format for storing graphs. Interestingly, the `Graph` package also contains data structures. There is a `Set` class, but more importantly, there is a `NodeList` class, which apparently represents a homogenous collection of nodes. There is no corresponding class for edges. The existence of custom data structure classes is acceptable because `VGJ` was written in `JAVA 1`, which did not provide a robust collections library.

In considering references to other classes, we see that `Graph` only refers to the classes in the same package, and to `DPoint3` from the `utils` package. The use of `DPoint3` suggests that graph elements can be associated with three-dimensional coordinates, so that spatial layout algorithms can be used.

We now turn to examining the actual `Graph` class. Surprisingly, although a graph is essentially a collection of nodes and edges, the class sports as many as 42 **public** methods. We shall therefore use `FCA` in order to analyze this large interface.

## 8.2.2 Constructing and simplifying the lattice

We construct a concept lattice of the interface of `Graph`, using the same selections we made in Chapter 5 (Fig. 5.2). The resulting lattice is depicted in Fig. 8.1.

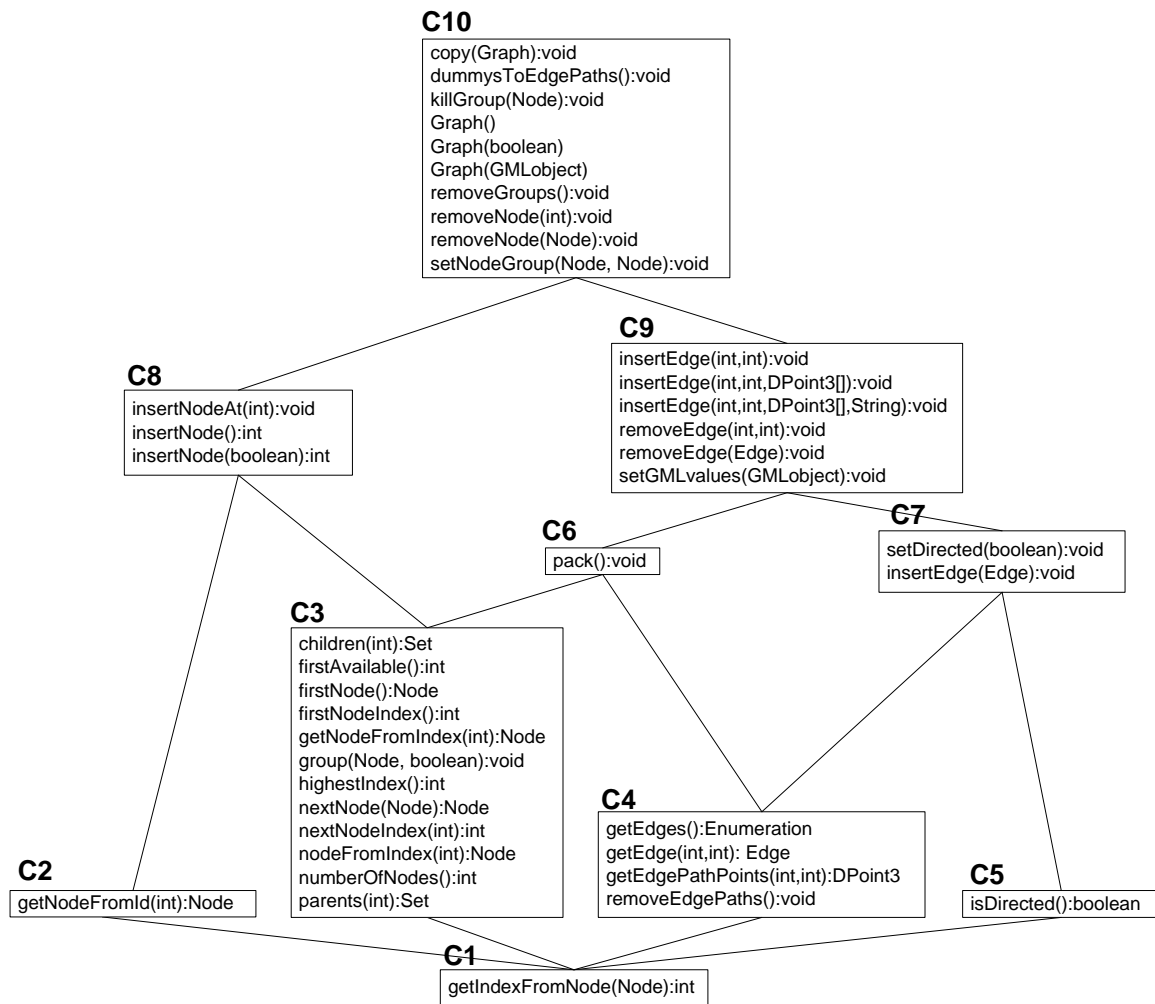


Figure 8.1: Concept lattice of `Graph` with full signatures and no fields

As we see in Fig. 8.1, the concept lattice partitions the 42 **public** methods of `Graph` into 10 concepts. In fact, as many as 36 of these methods are in 5 of the concepts, thus revealing strong similarities between the methods of the class. Because of the size of these large concepts, we postpone examining their methods in depth and instead create the summary lattice, depicted in Fig. 8.2.

The summary lattice in Fig. 8.2 shows that each of the large concepts contains several independent (although somewhat related) responsibilities which cannot be consolidated. This leads to a cumbersome naming of some concepts in the outline lattice which appears in Fig. 8.3.

The concept lattice of `Graph` is not horizontally decomposable. It does, however, sport only two top-level concepts, allowing us to group its concepts into four clusters in an abstraction lattice, as can be seen in Fig. 8.4.

Now that we have the pyramid of abstractions in the form of several lattices, we can start analyzing the interface of the class in depth.

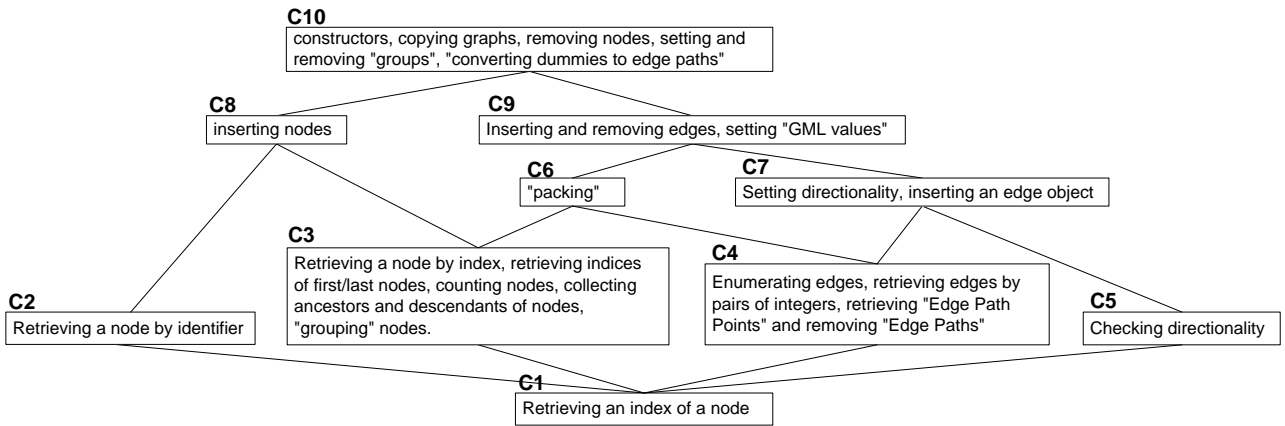


Figure 8.2: Summary concept lattice of Graph

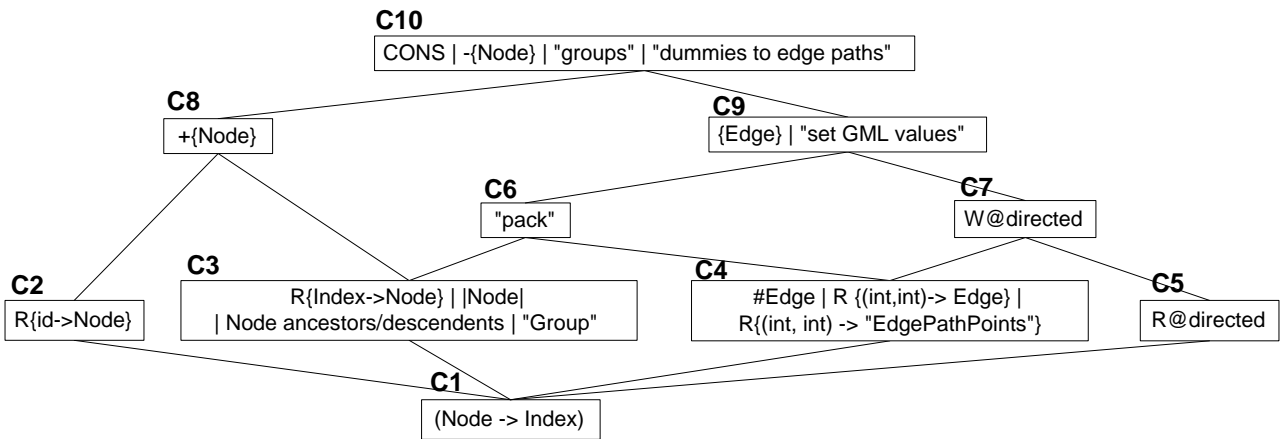


Figure 8.3: Outline concept lattice of Graph

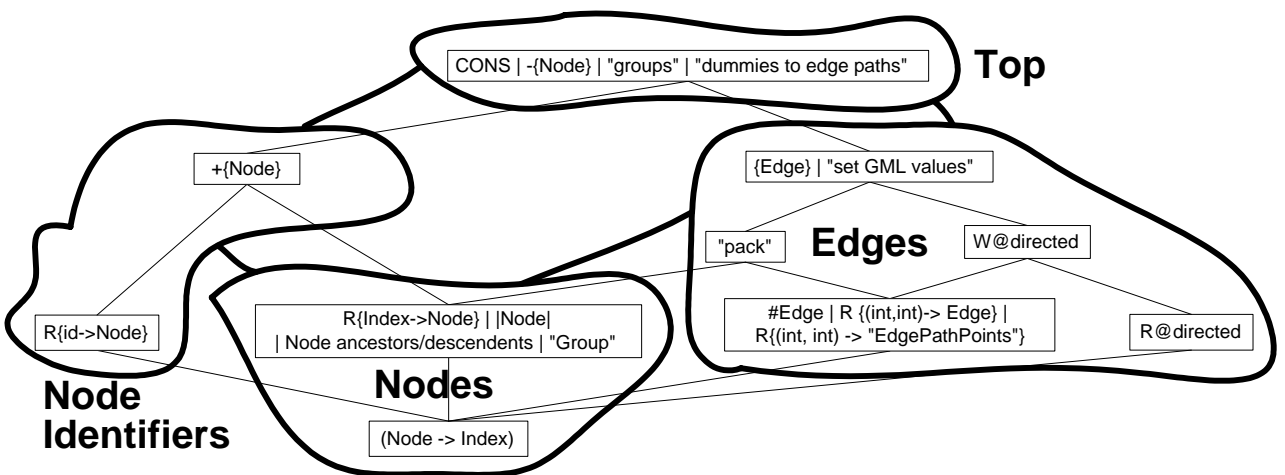


Figure 8.4: Abstraction lattice of Graph

### 8.2.3 Studying the lattices

As expected, the notions of nodes and edges appear in abundance in the interface of `Graph`. The abstraction lattice in Fig. 8.4 shows that the interface is clearly divided between methods dealing with nodes (the `Nodes` and `Node Identifiers` clusters), methods dealing with edges, and some methods dealing with both (in the `Top` cluster). Nevertheless, operations that deal with the same abstraction, such as adding and removing nodes, are separated between different concepts and even between clusters. The reason for this will be investigated when we study the implementation.

#### Nodes

We begin by scrutinizing the clusters dealing with nodes. The outline lattice shows that nodes are accessed in two distinct ways: by an identifier ( $R\{Index \rightarrow Id\}$ ) and by an index ( $R\{Index \rightarrow Node\}$ ). This redundancy is not only questionable, it also exposes implementation details by suggesting that nodes might be stored in an indexed collection. In other words, there appears to be a full-order between the nodes, which conflicts with the unordered nature of a general graph. The use of an identifier for a node is acceptable because it does not apply such an ordering.

We notice even more problems when examining concept  $C_3$  in the full lattice (Fig. 8.1). This concept contains 12 methods (more than a quarter of all the methods of the class), of which many appear to expose implementation details and should be replaced. The methods `firstNode`, `firstNodeIndex`, `highestIndex`, `nextNode`, and `nextNodeIndex` are based on the questionable full order between nodes. In fact, they appear to be useful mostly for iterating over the nodes; a method returning some iterator or enumerator would be more appropriate here. Also, the `firstAvailable` method suggests that clients might have to maintain and supply indices themselves; we make a mental note to investigate this during code inspection.

Another problem in concept  $C_3$  is the existence of both `nodeFromIndex` and `getNodeFromIndex`. Their names raise the suspicion that they perform the same function, so we write them down for further examination in the code inspection stage.

Finally, we notice that the node removal operations do not appear in the `Nodes`, but actually in the `Top` cluster. This suggests that unlike the `Molecule` class from CDK, where the removal of atoms did not cause the removal of the attached bonds, the removal of nodes in `Graph` correctly removes the incident edges.

#### Edges cluster

Turning to the edges cluster, the outline lattice shows that the `Graph` class can represent both directed and undirected graphs, and that it is possible to change the directionality of an existing graph instance. In addition, it seems that the class can only represent *simple graphs* (graphs without parallel edges), posing strict limitations on the possible uses of the class. We surmise this restriction from the `getEdge(int n1, int n2)` method in  $C_4$ : Since this method receives two integers which represent nodes, each edge can apparently be uniquely identified by the pair of nodes it connects and hence there can be no parallel edges.

The appearance of the operations dealing with edges in  $C_9$ , which dominates concepts dealing with nodes, edges and directionality, hints at some properties of these operations. For example, it is

possible that the removal of an `Edge` instance is checked against the existing graph to ensure that the edge indeed belongs to that graph.

The `pack` method in  $C_6$  suggests that the user must perform some maintenance on the class. The documentation of this method, “*Re-index so the indexes go from 0 to number of nodes - 1*” confirms that an implementation detail is exposed here. We make a mental note to investigate later whether this maintenance is mandatory or whether the class can function without it.

## Top cluster

The top cluster consists only of the top concept which, as expected, contains the constructors of `graph` and its `copy` method. It also contains methods for removing nodes, suggesting that the removal of a node causes the removal of adjacent edges. Finally, we encounter again the unfamiliar terms of “group” and “edge path”.

## 8.3 Stage II - Analyzing the implementation

### 8.3.1 Investigating fields

Before we reveal the fields of the `Graph` class that were hidden in the first stage, we are going to see that it is possible to surmise these fields solely from the outline lattice of the interface which appeared in Fig. 8.3.

We know that nodes can be accessed either by index ( $C_3$ ) or by identifier ( $C_2$ ). Since the methods for accomplishing this are separated among two concepts, we surmise that each of these concepts introduces a field which serves as a collection of nodes. The first is likely to be in the form of an array or a vector, and the second as some mapping or hash table.

One way to represent a graph is to have each node keep track of its incident edges, without maintaining a separate collection of all the edges. The fact that the methods for calculating the antecedents and descendants of a node in a directed graph (`children` and `parents` in  $C_3$  in Fig. 8.1) reside with methods that use solely a collection of nodes, seems to suggest that this is indeed the chosen representation. However, since the methods for dealing with edges appear in a separate concept ( $C_4$ ) then there is apparently a separate collection of edges, as a mapping keyed by a pair of node indices. It is not clear why an additional collection was used here since it certainly incurs more consistency-maintenance effort (such as the `pack` method which uses both collections). The only performance advantage of this approach seems to be in the case of a dense graph, where retrieving one node and then searching through its neighboring nodes is more expensive than searching in a mapping keyed by a pair of node indices.

Concept  $C_5$  which contains the `isDirected` method suggests that the class uses a boolean field to maintain whether the graph is directed or not. Yet, the directionality apparently has some effect on the state of the edges since `setDirected` appears in  $C_7$  which also dominates the edges collection.

We now turn to investigating the actual fields, and consult the lattice in Fig. 8.5 which includes both fields and non-public methods.

The fields in the bottom layer of the lattice confirm our expectations: edges are stored in a stan-

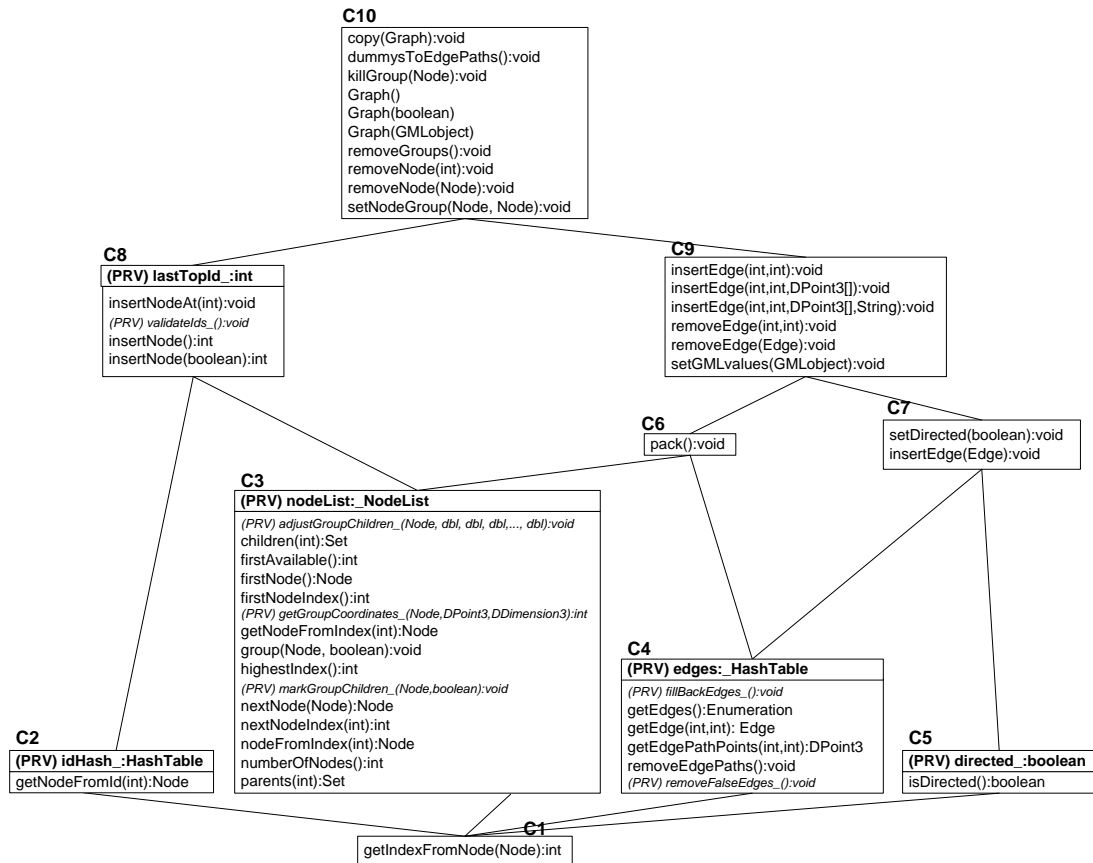


Figure 8.5: Concept lattice of Graph with non-public members

standard `HashTable`<sup>1</sup> field named `edges_`, and a boolean field named `isDirected_` flags directionality. Nodes are stored in two collections: field `nodeList_` is an indexed collection based on the special-purpose `NodeList` class, and `idHash_` is a mapping from identifiers to nodes, using the standard `HashTable` class.

In addition to the four fields of the bottom layer, a fifth field, `lastTopId:int` appears in concept  $C_8$ , and is therefore used only in conjunction with the `nodeList_` and `idHash_` fields. From its name and location we surmise that it is used to supply a unique identifier to each newly-inserted node.

We note that the name of the `idHash_` field is confusing since the field represents a mapping from identifiers to nodes. Hence, the name should represent its purpose, for example `mapIdsToNodes`, and not its implementation (a hash table of identifiers).

The existence of two collections for the same nodes requires them to be consistent, and we need to check that this is done correctly. For example, we need to ensure that operations that add or remove nodes affect both collections, and this is likely since these operations appear in concepts  $C_8$  and  $C_{10}$ .

It is also interesting that the `setGMLvalues` method, used to create a GML-based representation of the graph, is located in concept  $C_9$  which does not dominate concept  $C_2$ . Since concept  $C_2$  is likely to contain an identifier-based mapping of nodes, this method suggests that the mapping is redundant and perhaps used only to boost performance or usability. The indexed collection is stored in the file and can later be used to recalculate the mapping when that file is loaded (we have a constructor that accepts a `GMLobject` in the top concept).

### 8.3.2 Investigating Methods

#### Methods dealing with the entire state or no part of the state

As directed by our methodology, we first deal with methods that use the entire state and those that do not use any part of the state. In addition to the constructors and the `copy` method, the top concept contains methods for node removal and several additional methods. Since node removal uses both the `edges_` and `directed_` fields, we surmise that when a node is removed, all incident edges are removed as well. As we can see in the restricted ECG of the top concept in Fig. 8.6, the other methods in that concept use the node removal method and therefore use the entire state as well.

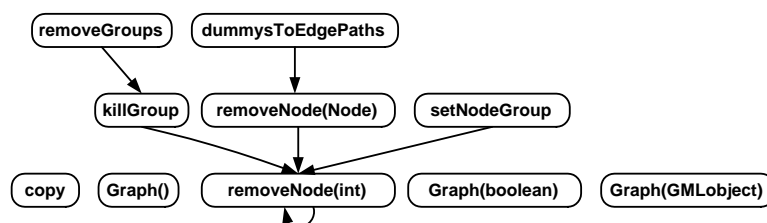


Figure 8.6: Embedded call graph of `Graph` restricted to the top concept

The bottom concept of the graph contains the `getIndexFromNode(Node):int` method. The fact that this method does not use the `nodeList` field suggests that each `Node` instance maintains

<sup>1</sup>VGJ was developed before the release of `JAVA 2`, where the `Map` interface replaced `Hashtable`.

its own index in the list. If this questionable behavior is indeed the case, then this method is a simple delegator and it is not clear why it is part of the interface of `Graph`. Furthermore, the method is not **static**, which is apparently an error by the developer.

### Studying asymmetries

We already saw a clear asymmetry between the representation of nodes and edges: Whereas edges are stored in a single collection, the nodes are represented using two collection fields and an additional integer. The difference between the representation of the nodes and the representation of edges is very clearly manifested in the interface presented to the user and hence in the way clients use the `Graph` class.

One problem is that a user can access nodes either by index or by identifier. This redundancy is confusing to users, especially since both kinds of accesses use integers. For example, a user might accidentally retrieve an edge using two identifiers instead of two indices. Although the use of identifiers is more natural because graphs are unordered, we saw that this field is actually the redundant one in the current implementation of the class.

Another asymmetry between nodes and edges is that more wrappers and non-primary methods are available for nodes than for edges. For example, there is a special method for retrieving the number of nodes in a graph, but no such method is provided for edges. On the other hand, it is simpler to enumerate edges using the standard mechanism than to enumerates nodes using the methods provided in concept  $C_3$ .

### Non-public members

We now examine the non-public methods of class `Graph`, which appear in *italics* and are prefixed by `PRV` in Fig. 8.5. There are six **private** methods altogether: `validateIds_` in  $C_8$ , which is used for bookkeeping node identifiers, methods dealing with “group children” in  $C_3$ , and two methods in  $C_4$  which prove valuable to our investigation: `fillBackEdges_` and `removeFalseEdges_`. These two methods suggest that there are different types of edges in the system (we already know that there are different types of nodes). We consult the ECG of `Graph` (Fig. 8.7) and learn that both methods are used only by `setDirected`.

The names of the `fillBackEdges_` and `removeFalseEdges_` methods, along with the fact that both are invoked by `setDirected_` hints at how the `Graph` class deals with edge directionality. Apparently, each `Edge` object is directed, and undirected graphs are emulated using additional dummy edges, that are added and removed when the directionality of the graph changes. We will attempt to verify this assumption when we inspect the code in the next stage, and will also try to find out why two different terms, back-edges and false-edges are used.

## 8.4 Stage III - Inspecting the code

As dictated by our methodology, the inspection of the code begins with the bottom concept and ends with the top concept. In the interest of conciseness, we demonstrate here only the results of inspecting the bottom layer, whose concepts are read from the simplest to the most complex.

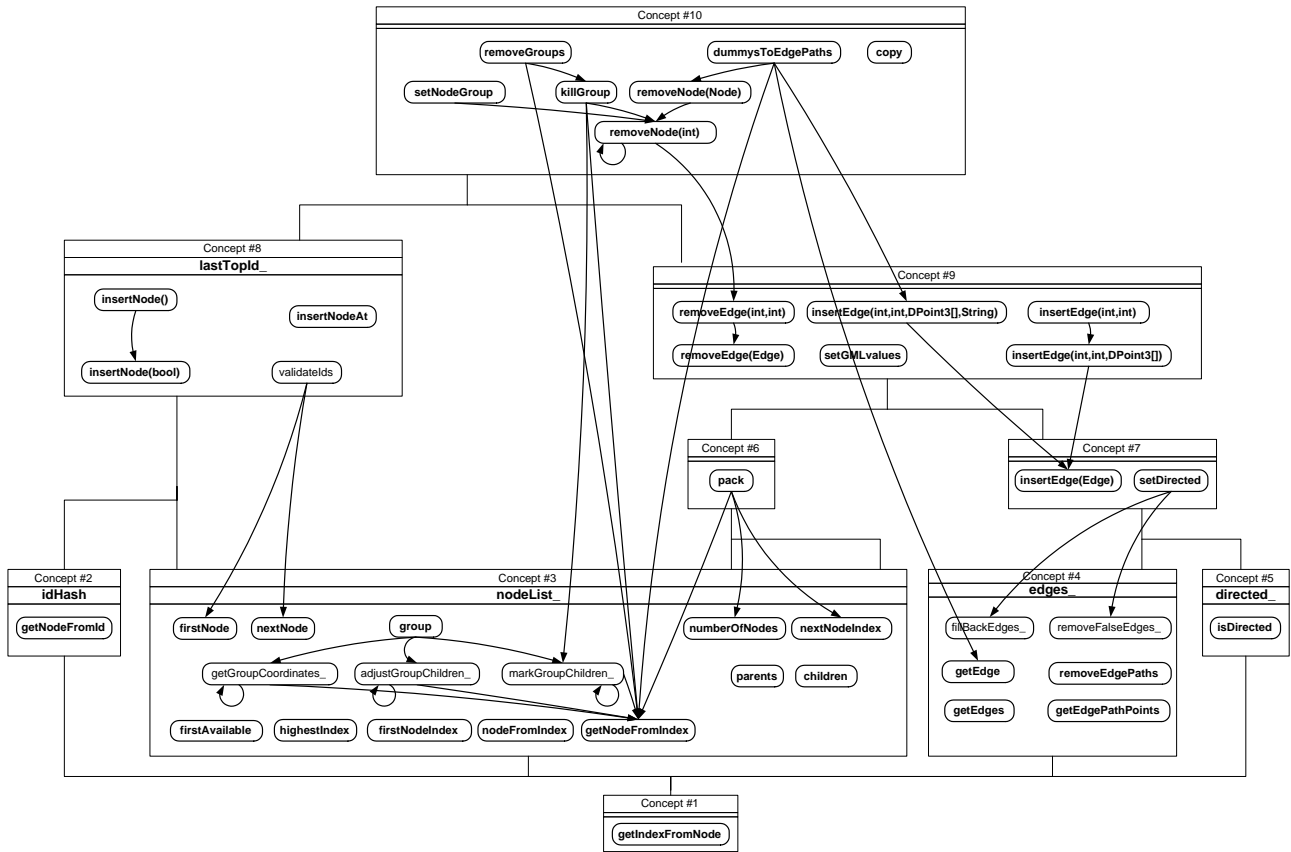


Figure 8.7: Embedded call graph of Graph

Concepts  $C_2$  and  $C_4$  contain only one method each. The first contains `getNodeFromId` which simply delegates to the hash table, and concept  $C_2$  contains only the trivial `isDirected` method, which (surprisingly) contains redundant code.

Concept  $C_5$  contains several methods, which we examine in an ascending order of simplicity. The `getEdges` method simply returns an enumeration of the edges. We make an automatic search for code sections that bypass this method and indeed find many occurrences, which can be fixed easily with a simple search-and-replace. Next we examine `getEdge`, which accepts two node indices. In its code we see that a `Point` object is created from these indices to serve as a key for the hash table. Since the `Point` class belongs to the graphical AWT package and has many features, its use here is inappropriate. A more economical `Pair` class should be defined and used here and in any other method that deals with the edges collection.

The code of `fillBackEdges_` and `removeFalseEdges_` in  $C_5$  confirms our earlier assumption that undirected graphs are emulated using additional edges, and that the two terms actually refer to the same thing. In addition, we learn that the additional edges are not specially flagged, and as a result the simulation is not complete. Suppose that a user attempts to count the nodes of an undirected graph by checking the size of the enumeration. The resulting size would be twice the actual value! The methods in concept  $C_5$  should be redesigned to better hide these implementation details from clients, for instance by filtering the enumerations of edges.

Another interesting problem here is that the newly created edges are not identical to the ones that they “mirror”. Suppose that a user subclasses or adds information to an edge, and later attempts to retrieve that edge with juxtaposed indices (which in an undirected graph should be equivalent). The retrieved edge would then be a different object from the one to which the data was added. In fact, this problem will occur if the client uses the version `insertEdge` method that accepts four parameters (in  $C_9$ ). The last parameter is a string label, which is only placed in one copy of the edge. If the mirrored edge is retrieved, the label will be incorrect.

Concept  $C_2$  is the largest concept in the bottom layer (and in the lattice), with 12 **public** and 3 **private** methods. The ECG shows that the **private** methods are used only by `group` and will be read along with it. To select an order between the **public** methods, we use a static analyzer to collect metric values for these methods. These values are listed in Table 8.1, which partitions the methods into five groups according to the metric values.

The first two groups in Table 8.1 contain nine methods that simply delegate to the `nodeList` field<sup>2</sup>. The interface of the `NodeList` class, to which they delegate, only contains 13 methods<sup>3</sup>. Hence, most of the interface of `NodeList` is replicated in `Graph` and significantly clutters its interface. Since these methods are mostly used for iterating over nodes, they should be replaced by a standard enumerator.

A certain evidence that the nine delegators clutter the interface of `Graph` is that two of them, `nodeFromIndex` and `getNodeFromIndex` actually perform the same function. This redundancy which was missed by the original developer (perhaps since the methods are not adjacent) became obvious when we examined the lattice of the interface and is confirmed now that the code is examined.

---

<sup>2</sup>The difference in bytecode length arises from the fact that the methods in the second group must delegate their parameters.

<sup>3</sup>Excluding constructors and cloners.

Group	Name	BClen <sup>a</sup>	MCC <sup>b</sup>	Nmsg <sup>c</sup>	Nacc <sup>d</sup>
1	highestIndex():int	8	1	1	1
	firstAvailable():int	8	1	1	1
	firstNode():Node	8	1	1	1
	firstNodeIndex():int	8	1	1	1
	numberOfNodes():int	8	1	1	1
2	getNodeFromIndex(int):Node	9	1	1	1
	nextNode(Node):Node	9	1	1	1
	nextNodeIndex(int):int	9	1	1	1
	nodeFromIndex(int):Node	9	1	1	1
3	children(int):Set	12	1	2	1
4	parents(int):Set	50	3	5	2
5	group(Node, boolean):void	306	5	12	17

<sup>a</sup>Bytecode length

<sup>b</sup>McCabe's Cyclomatic Complexity

<sup>c</sup>Number of invoked methods

<sup>d</sup>Number of accessed fields

Table 8.1: Metrics for the methods of concept 2 in the lattice of Graph

Because the nine delegators are linear, it is possible that they are bypassed in other methods. This is more severe than bypassing field-get and field-set methods because it creates a coupling with the class of the field. Indeed, an automatic search discovers multiple places where these fields are bypassed.

The differences in size and complexity between the `children` and `parents` methods hints at their implementation and time complexity. Apparently, each node keeps the set of its descendants (reachable neighbors), allowing them to be retrieved immediately. The set of antecedents (reaching neighbors), on the other hand, must be calculated in `parents`. An examination of the code confirms this, as the parents of a node are calculated by iterating over all the nodes. It is also not clear why the code of `parents` does not optimize for the case of an undirected graph, where `children` is supposedly equivalent.

# Chapter 9

## Conclusions and Future Research

In this work, we proposed a novel approach to the understanding and analysis of individual classes, and demonstrated that FCA can be useful when applied to the limited scope of a single class. The underlying assumption was that the use of fields by methods can be used to classify methods into meaningful groups and discover interface and implementation problems that might be obscured when carrying out a code inspection that focuses on the coding details of individual methods.

In addition, this work proposed a systematic methodology for the study of classes based on the use of concept lattices, and demonstrated it on the two case studies presented in the previous chapters. In order to confirm the effectiveness of this methodology compared to other analysis and inspection methodologies, an empirical user study is required.

Although many user studies on inspection techniques appear in the literature (e.g. [14]), most are concerned with entire systems and with an examination of either the requirements or code. This makes it possible to devise studies that compare the effectiveness of the code inspection technique which we proposed in Chapter 7 against other techniques or against an unguided code reading. Unfortunately, such candidates for comparison are not available, to our best knowledge, when it comes to asserting the usefulness of the first two stages of our methodology, which deal with understanding the class without examining the code of its methods. We therefore leave it for future research to design an experiment that can reliably evaluate the ability of programmers to understand a class and discover problems based only on its interface and access patterns.

Before a large-scale user study can be conducted, the automatic and semi-automatic tools discussed in this work must be provided as an interactive software product. We currently have a command-line based prototype that takes a class file and produces a variety of lattices which correspond to some of the zoom-in and zoom-out tools. Such a mode of operation is not sufficient for a user study and for commonplace use, and more interactive tools are needed. We are currently developing such a tool which will be deployed as a *plugin* for the open-source *eclipse* development environment [15].

The following sections discuss additional research directions that build upon the foundation laid in this work.

## 9.1 A lattices based metrics suite

Software metrics are an active research area, and much work deals with the development of new metric suites for object oriented systems [11], classes, and methods [12].

We suggest developing a metrics suite for classes based upon properties of their lattices. This idea follows from the seminal work of Lanza and Ducasse [37] on *class blueprints*, which demonstrated that much can be learned about a class from observing an outline of its structure, even when actual names of members are not considered. Because concept lattices are based on the structure of the class, it might be possible to predict the properties of that class using the values of metrics of its lattice. In addition, these values might be useful in estimating the applicability of concept analysis for a certain class, and for selecting a subcontext that reveals as much information as possible while incurring a minimal loss of information.

We now describe some possible metrics:

**Number of objects and number of attributes** Correlating to the *number of fields* and *number of methods* class metrics, these (and especially their ratio) give a rough estimate to the size of the class and its structure. For example, in a **struct**-like class, there would be two attributes (a getter and a setter) for every field.

**Number of concepts** The actual number of concepts, especially when compared to the maximal possible number of possible concepts, gives a rough idea as to the similarity or lack thereof between methods. On one extreme, we have fully-cohesive classes where all methods use all fields, and on the other extreme we have complex classes where every method uses a different combination of fields. Further information can be inferred by classifying concepts into *abstract* (empty) concepts, *concrete* concepts (with nonempty sets of objects and attributes), and *connector* and *intersect* concepts (with empty sets of objects or attributes, respectively).

**Number of entries in the context** The cardinality of the context's relation (the number of checked cells in the table) and its percentage from the maximal possible cardinality can be calculated from the lattice. In highly cohesive classes, this number will be high. In a **struct**-like class, this number will be linear in the number of fields.

**Height** Because layers of the lattice often correspond to levels of abstraction, the height of the lattice may correlate to the number of such levels in the class.

**Average height of fields and methods** In well-written classes, we expect to see fields introduced only in the bottom layer, and to see the number of introduced methods significantly decrease as we ascend in the lattice. The average heights where fields and methods are introduced in the lattice can tell us whether these expectations hold, especially when accompanied by other statistical metrics such as standard derivatives.

**Total number of edges, number of children and parents per concept** The number of edges can serve as an indicator to the complexity of a class. In a string-like or tree-like lattice, this number is low, whereas in a complex lattice with many interferences this number is high. The average numbers of children and parents per concept are also interesting.

**Shape** The shape of the lattice (when top- and bottom- concepts are removed) is a good indicator of the nature of the class. For example, a **struct**-like class would form a wide single-layer line, whereas classes that have a clear distinction into services and levels of abstraction might have

the shape of a tree. We can assign a numeric magnitude to each shape, to represent the level of class organization that it indicates.

**Entropy** Lattices of classes are useful when they successfully partition methods into nontrivial groups (i.e., with more than one concept), as long as these concepts are not too large. In order to convey the usefulness of FCA for a particular class or aid in selecting a subcontext, it is vital to develop a metric that measures the amount of information conveyed by the lattice. Such a metric is likely to be based on Shannon's *entropy* [46], but will have to take into account the special properties and limitations of concept lattices.

Research on developing such a metric suite would involve defining metric values and calculating them for a large sample. The resulting values would be analyzed and the suite improved to eliminate correlations and dependencies. Then, *self-calibration* [12] can be used to ascertain a *common programming practices value*, and possibly identify implications between metric values and class properties.

## 9.2 Interactive design of classes

Although this work dealt only with the use of concept lattices for the analysis of existing classes, we believe that this technique has potential for the interactive design of new classes and the modification of existing ones.

Consider how methods and fields are added to a new class in most CASE tools or development environments. These members are added to a list of existing members which appears in a "class viewer" window of an IDE or in the UML *class diagram* [56] of a CASE tool. As this list, which is typically sorted in a trivial lexical order, grows to contain a large number of methods, adding new methods and ensuring that they are not already available in another name becomes tedious. Another problem with this methodology is that most of the methods in the list are left without any semantics during the method-addition stage. The reason for this is that the only way to associate semantics is to add documentation or actual code, which is time consuming and interrupts the flow of method additions. Only after enough methods have been added, does the programmer start providing the actual code, doing so without having any indications (aside from the method names) as to the expected semantics.

To alleviate the problems mentioned above, we suggest a new kind of class editor for development environments and CASE tools. The main feature of this editor is an interactive concept lattice. The editor enables us to add new methods and fields like the existing editors, but also to associate methods with fields. The concept lattice reflects these associations, and as a result displays related methods together. New methods can then be added directly to existing concepts.

The advantage of this approach is that it presents the methods in meaningful groups (using the same heuristic we used for analyzing classes), so that it is easier for the developer to locate a certain method or to check if it was already provided. In addition, it allows us to assign initial semantics to methods.

The design tool described here can be further supplemented by supporting the automatic application of *nano-patterns* [23], short linear methods that recur in many programs. Examples of such patterns are field getters and setters, delegators, etc.

# Appendix A

## Previous Applications of Concept Analysis in Software Engineering

The introduction of *formal concept analysis* (FCA) as a conceptual clustering technique by Wille [60] was followed by many works dealing with the theory of concept analysis, and by works suggesting applications of the technique to problems in various fields. In this chapter we survey the works on applications of FCA to software engineering. A reader who is not familiar with the theory of concept analysis is encouraged to refer to Chapter 3, or to the short primer on concept analysis which appears in [48]. Also available are books (e.g. [22]) that treat the theoretical aspects of this technique in depth.

Although there are many applications of concept analysis in software research, most of the research falls into one of the following research directions:

**Modularization** Modules in procedural languages are supposed to be highly cohesive units, but many legacy programs do not have a correct explicit modular structure due to language restrictions, poor design, or maintenance problems. A concept lattice can be used to discover cohesive units and suggest a decomposition of the program into modules. Some previous contributions attempt to identify abstract data types which can become objects in a transition to an object oriented language. An additional problem which we include in this category is the maintenance of multiple configurations, where code for different target executables is mixed in the same source file.

**Class hierarchies** In one of the early papers on applying concept analysis to software research, Godin and Mili [26] suggested a methodology for automatically constructing a class hierarchy. Other works propose more strategies of constructing new hierarchies, and re-engineering existing ones. The automatic construction of hierarchies of interfaces is also discussed.

**Component retrieval** Reuse of existing software component is a highly sought goal in software development because of the economical benefits it entails. Even if high-quality reusable components exist in the software repository of an organization, effective retrieval techniques are still needed to bring them the attention of developers. Several works apply concept analysis to the exploration and search of software component repositories, and consider the stored software components to be objects with certain sets of properties.

**Reverse engineering** Reverse engineering of legacy systems written in procedural or object oriented languages entails many benefits and serves as a first step towards program understanding.

Works dealing with reverse engineering exploit the classification of information into concepts to recover different elements or formal models from the implementation or architecture of the program.

**Program comprehension** The distinction between reverse engineering and program comprehension is vague because both objectives often coexist. Under this category, we classified works that aim at improving the understanding of the system without obtaining formal models.

The remainder of this chapter discusses each category in depth. Table A.1 summarizes the works we shall discuss by category.

Category	Works
Modularization	<i>Assessing modular structure of legacy code based on math. concept analysis</i> [39]. <i>Concept analysis - a new framework for program understanding</i> [50]. <i>Identifying modules via concept analysis</i> [48]. <i>Concept analysis for module restructuring</i> [54]. <i>Identifying objects using cluster and concept analysis</i> [57]. <i>Types and concept analysis for legacy systems</i> [35].
Class Hierarchies	<i>Building and maintaining analysis-level class hierarchies using Galois lattices</i> [26]. <i>Understanding class hierarchies using concept analysis</i> [51]. <i>Computing interfaces in Java</i> [27].
Component-Retrieval	<i>Concept-based component retrieval</i> [38]. <i>Concept-based retrieval of classes using access behavior of methods</i> [47].
Reverse-Engineering	<i>Object oriented design pattern inference</i> [55]. <i>A use-case driven method of arch. rec. for prog. understanding and reuse reeng.</i> [9]. <i>A method to reorganize legacy systems via concept analysis</i> [1]. <i>Architectural element matching using concept analysis</i> [58].
Program-Comprehension	<i>Aiding program comprehension by static and dynamic feature analysis</i> [16]. <i>Feature-driven program understanding using concept analysis of execution traces</i> [18]. <i>Derivation of feature component maps by means of concept analysis</i> [17].

Table A.1: Works on applications of concept analysis, by category

## A.1 Modularization

Lindig and Snelting [39, 50] showed that a concept lattice of the relation between procedures and global variables in legacy code (written in FORTRAN and COBOL) reveals potential module candidates. If the program has an inherent modular structure, the resulting lattice should be *horizontally decomposable* (Definition 13 in Chapter 5). Occasional couplings between modules appear as *interferences*, concepts serving as infima for concepts of different horizontal summands. If the number of interferences is small, various *interference resolution* techniques can be applied. If there are too many interferences and no clear modular structure exist, various decomposition techniques based on FCA theory are used in an attempt to discover some structure.

In the conclusions of their paper, Lindig and Snelting [39] argue:

Basic mathematical concept analysis, as used in this article, is not “continuous”: a single “wrong” entry in the variable usage table can destroy decomposition properties of the lattice. This behavior seems to prevent automatic modularization in many cases. A more realistic restructuring approach must probably include some heuristics.

Siff and Reps [48] propose the use of other attributes to alleviate this problem. They use contexts which relate program functions not only to fields of user defined structures and fields, but also to the data types of expressions that occur inside these functions. These expressions include returned values, arguments, information from *type inferencing*<sup>1</sup>, etc. In addition, they show that the use of *negative information* as attributes, e.g. “function  $f$  does *not* return a value of type  $t$ ”, can further simplify the resulting lattices. Finally, they define the notion of *concept partition*, a set of concepts which partitions the entire set of objects (program functions). Each of the possible concept partitions of the lattice represents a possible modularization, and therefore the lattice structures represents all the possible modularizations of the class.

The notion of concept partitions for modularization is further explored by Tonella [54], who argues that the inherent requirement from concept partitions to cover the entire set of objects is too restrictive and results in a loss of important information. Instead, he introduces a weaker notion of *concept subpartitions*, which does not have to cover the entire set of objects.

Tonella also discusses the problem of balancing encapsulation and cohesion in modularization, arguing that the two are often conflicting, and presents metrics for estimating them. In addition, he compares modularization by concept lattices to modularization by *clustering*<sup>2</sup>. He argues that clustering essentially builds module candidates according to cohesion and coupling metrics as it measures the distance between items, whereas concept analysis structures modules by semantics. He concludes that the advantage of using FCA is that concepts can be interpreted as describing certain common properties of entities, whereas there is no direct interpretation for clusters.

Van-Deursen and Kuipers [57] who also consider cluster and concept analysis for modularization (with a purpose of identifying potential classes) compare them on a case study and conclude that “concept analysis is more suitable for object identification than cluster analysis”. They mention the following benefits:

- A concept lattice presents all the possible groupings whereas a clustering represents just one partition.
- It is easier to trace a concept to the common attributes of its objects (the same conclusion reached by Tonella).
- Cluster analysis is over-sensitive to items that possess all features, and requires special processing.

Kuipers and Moonen [35] discuss the problem of identifying types in COBOL, where only primitive types exist. They start by applying a type-inferencing technique they developed, and use the inferred types for their analysis. According to their empirical results, the methodology yields a higher precision than that which can be obtained by a simple use of variables and records. In one experiment, they use inferred types as objects and COBOL programs as attributes, obtaining concepts that

---

<sup>1</sup>Type inferencing attempts to discover logical types such as enumerations when more general types such as integers are used.

<sup>2</sup>A process known as *cluster analysis*, which uses a metric of the distance between items to create cohesive clusters.

represent collections of related programs. In a second experiment, the programs are objects, and the types of formal parameters are attributes. The resulting concepts in this experiment are sets of programs sharing the same formal arguments, making them candidates for transition to a class where each program is a method.

Kuipers and Moonen also discuss the idea of refining concepts in order to obtain a simpler lattice. Using a lattice browsing application they developed, a user can combine or remove table columns and rows, or combine lattice concepts. In Chapter 5, we discuss an automatic refinement which is useful for the concept lattices we use.

Another application of concept analysis which can be considered as modularization is the problem of maintaining multiple configurations. A source file can contain code which is intended for several platforms, with slight differences. In C, the code sections that are actually compiled are determined by the values of preprocessor variables and expressions.

For programs where only preprocessor variables are used, Snelting [50] applies concept analysis by using pieces of codes that appear under the same preprocessor expression as objects, and using preprocessor variables as values. Each concept represents a configuration thread, and the lattice structure improves the understanding of the different threads and occasionally helps discover dead code.

## A.2 Class hierarchies

One of the first applications of concept analysis for software engineering was the work of Godin and Mili [26] on the automatic construction of class hierarchies. The authors demonstrated that a concept lattice can be used to obtain a hierarchy of related classes which need support different protocol elements. In their example, a set of SMALLTALK message names were used as objects, and the names of different collections were used as attributes. The context relation specified for each message the collection classes which should respond to it. The resulting lattice was interpreted as a class hierarchy, and following several transformations could be used in a single-inheritance environment. Finally, they discuss the construction of hierarchies using more elaborate class descriptions and properties.

Snelting and Tip [51] took a different approach to the engineering of class hierarchies. They argue that the actual use of classes by user programs can help improve the hierarchy. For example, if each user program uses a different subset of the members of a class, it is possible that the class is too general and should be specialized by several subclasses. Their methodology takes variables and pointers to class instances as objects, declarations and definitions of class members as attributes, and the use or access of class members through the variables as a context relation. The resulting lattice, following several transformations, can represent a better class hierarchy which can be used for re-engineering or for a quality assessment of the existing hierarchy.

Huchard and Leblanc [27] use a variant of concept lattices, called a *Galois sub-hierarchy* to compute a multiple-inheritance hierarchy of interfaces in JAVA from a single-inheritance class hierarchy. Each class in the hierarchy is considered as an object, and each method it supports as an attribute. FCA is used to calculate a *Galois sub-hierarchy*, which can be thought of as a lattice from which all empty concepts (including the bottom- and top- ones) are removed. This sub-hierarchy represents a hierarchy of interfaces, which the original classes should implement.

## A.3 Component retrieval

Lindig [38] presented a software prototype for retrieving reusable components from a repository. The repository is represented in the form of a concept lattice, where objects represent software components and the attributes are keywords describing these components, taken from the documentation. The concept in which a certain keyword appears dominates all the concepts which contain components with that keyword. Therefore, by incrementally selecting keywords from a list, the user can refine the set of matching components until a suitable component is found.

Shen and Park [47] use FCA to manage a repository of classes. The retrieval is based on the use of types by class methods, for example “*find classes with methods that read and write a variable of type X*”. Each object in their context is a pair of a type and an access pattern, and each attribute corresponds to a class in the library. The context relation specifies that a certain type is accessed in the specified pattern by at least one method in a class. Again, a query involves a search in the lattice until the appropriate class(es) are found.

## A.4 Reverse engineering

Tonella and Antoniol [55] propose an FCA-based approach to the problem of discovering design patterns in a system in the absence of a catalogue of patterns. For a fixed  $n$  (which is restricted to avoid combinatorial explosion), the objects in their context are ordered sequences of  $n$  classes. Each attribute is a pair of two class indices with an associated kind of relation (e.g., *extends*). The context relation then specifies for each sequence of classes the relations between the members of the sequence. The rationale behind this approach is that a recurring design pattern would be evident as a sequence of classes with the same interrelationship between them. Therefore, they only calculate the concepts (the lattice itself is not calculated), and take large concepts as candidates for patterns. The notable problem with this approach is combinatorial explosion, and the paper offers different approaches to ameliorating it.

Another common artifact of reverse engineering is a UML [56] diagram. Bojic and Valesovic [9] attempt to reverse engineer a program and obtain a UML diagram using FCA. Their context is based on user-defined use-cases and dynamic information from the execution of test cases that realize these use-cases. The objects are all the functions of the program (members and non-members), and the attributes are the names of user-defined use cases. The context relation specifies for each function,  $f$ , and each use case,  $u$ , whether  $f$  “implements”  $u$ : if at least one test case that executes  $f$  covers  $u$  and every test case that executes  $f$  covers  $u$ , then  $f$  implements  $u$ . The resulting lattice can be interpreted as a hierarchy of logical UML packages. The unique placement of items belonging in multiple packages (due to the multiple-inheritance nature of the concept lattice) is resolved using heuristics. The logical-package hierarchy serves as a starting point for discovering the elements inside each package.

If the previous work extracted a logical hierarchy of packages, we can think of the work of Antoniol et al. [1] as an attempt to extract a physical hierarchy of packages. Their work, dealing with legacy systems, attempts to reconstruct the directory tree of legacy applications whose structure was lost during evolution. They take the executables of the legacy system as objects and the source and object files as attributes. The relation specifies whether a certain file is required to link a certain

executable. The concepts of the resulting lattice represent candidates for being libraries, and their location in the lattice is interpreted as showing their “level of use”. In their experiments, they flattened existing programs and tried to construct a hierarchy (DFS was used to create a tree from the lattice). They claim that their system either created the original directory structure, or a different structure which they deemed to be of high quality.

One of the problems of reverse engineering and architectural recovery techniques is that each produces an output (called “perspective”) at a different level of granularity. For example, one tool may identify a system whereas another tool might identify its breakdown into subsystems. Waters et al. [58] use FCA in an attempt to combine different perspectives that results from different reverse-engineering techniques. Using a special technique for identifying recurring keywords in source and documentation, a set of domain related terms is gathered and used as attributes for the context. The objects of this context are architectural elements (such as components or connectors) associated with the perspective that identified them. The relation then specifies whether a particular element was identified as connected to a certain domain term. In the resulting concept lattice, each concept should consist of equivalent elements from different perspectives, and the lattice should exhibit the containment or granularity differences between the discovered items.

## A.5 Program comprehension

The distinction between program comprehension and reverse engineering is very vague. We consider here works that attempt to solve the *concept assignment problem*, or the *feature-component correspondence* problem, which are one of the important problems in program comprehension and reverse engineering. These problems deal with discovering what “human concepts” or features and requirements are realized by what software entities. Eistenbarth et al. [16] describe the problem as follows:

Understanding how a certain feature is implemented is a major problem of software understanding, especially when the understanding is directed to a certain goal like changing or extending the feature. Before real understanding starts, one has to localize the implementation of the feature in the code.... It is in general not obvious which components implement a given feature. Typically, any existing documentation is outdated, the system’s original architects are no longer available or their view is outdated to due to changes made by others.

The authors take an approach somewhat similar to that used by Bojic et al. [9] for obtaining a UML diagram from use-cases. They construct a context where objects are subprograms of legacy code and attributes are scenarios or execution profiles gathered using dynamic analysis. The context relation specifies the subprograms which are executed when each scenario is performed. The resulting lattice is used to learn about the relations between subprograms, and what irrelevant subprograms are activated in the performance of each scenario. This information is used to eliminate these items from the dependency graph for a subsequent static analysis of this graph.

The same authors apply variations of this technique in [18], where the attributes are the *features* (realized requirements) of the software, and the objects are program entities. These entities include procedures or subprograms, or higher-level entities such as modules and components when a physical

or logical modularization is available. The context relation specifies whether a certain program entity is used in a scenario that activates a certain feature. In the resulting concept lattice, each concept corresponds to a set of entities that are used to realize a particular feature. Another variation is used in [17] to investigate reuse in product lines by mapping features to components or products.

# Bibliography

- [1] G. Antoniol, G. Casazza, M. D. Penta, and E. Merlo. A method to reorganize legacy systems via concept analysis. In *Proceedings of the 9<sup>th</sup> International Workshop on Program Comprehension* [28], pages 281–291.
- [2] L. Barowski and C. McReary. Visualizing graphs with Java. Reverse engineered with permission.  
[http://www.eng.auburn.edu/department/cse/research/graph\\_drawing/graph\\_drawing.html](http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html).
- [3] L. A. Belady and M. M. Lehman. Laws of program evolution dynamics. *IBM Syst. J.*, 15(3):225–252, 1976.
- [4] A. Berry and A. Sigayret. Representing a concept lattice by a graph. In *Proceedings of the 2<sup>nd</sup> SIAM International Conference on Data Mining, Workshop on Discrete Mathematics and Data Mining*. SIAM, 2002.
- [5] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the the 17<sup>th</sup> international conference on software engineering, Symposium on software reusability*, pages 259–262. ACM Press, 1995.
- [6] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15<sup>th</sup> International Conference on Software Engineering*, pages 482–498. ICSE '93, IEEE Computer Society Press, 1993.
- [7] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, May 1994.
- [8] G. Birkhoff. *Lattice Theory*. Colloquium Publications. American Mathematical Society, Providence, RI, USA, 2<sup>nd</sup> edition, 1967.
- [9] D. Bojic and D. Velasevic. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 23–32, San Jose, CA, USA, Oct. 2000. ICSM '00, IEEE Computer Society Press.
- [10] Chemistry Development Kit (CDK) homepage. Reverse engineered with permission.  
<http://cdk.sourceforge.net>.
- [11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

- [12] T. Cohen and J. Y. Gil. Self-calibration of metrics of Java methods. In *Proceedings of the 26<sup>th</sup> International Conference on Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. TOOLS Pacific 2000, Prentice-Hall.
- [13] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of the 22<sup>nd</sup> International Conference on Software Engineering*, pages 467–476. ICSE '00, ACM Press, 2000.
- [14] A. Dunsmore, M. Roper, and M. Wood. Practical code inspection for object-oriented systems. In M. Lawford and D. Parnas, editors, *WISE'01: Proceedings of the 1<sup>st</sup> Workshop on Inspection in Software Engineering*, pages 49–57. Software Quality Research Lab, McMaster University, Hamilton, Canada, 2001.
- [15] Eclipse project homepage. <http://www.eclipse.org>.
- [16] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 602–611, Florence, Italy, Nov. 2001. ICSM '01, IEEE Computer Society Press.
- [17] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings of the 5<sup>th</sup> European Conference on Software Maintenance and Reengineering*, pages 176–179, Lisbon, Portugal, Mar. 2001. CSMR '01, IEEE Computer Society Press.
- [18] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the 9<sup>th</sup> International Workshop on Program Comprehension [28]*, pages 300–309.
- [19] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, 1976.
- [20] B. Fischer. Specification-based browsing of software component libraries. In *Proceedings of the 13<sup>th</sup> IEEE Conference on Automated Software Engineering*, pages 74–83, Honolulu, Hawaii, USA, 1998. ASE '98, IEEE Computer Society Press.
- [21] G. Funk, A. Lewien, and G. Snelting. Algorithms for concept lattice decomposition and their applications. Technical Report 95-09, TU Braunschweig, Dec. 1995.
- [22] B. Ganter and R. Wille. *Concept Analysis: Mathematical Foundations*. Springer, 1999.
- [23] Y. Gil and S. Porat. The simplicity of Java methods. In preparation.
- [24] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.
- [25] Gnu license. <http://www.gnu.org/copyleft/gpl.html>.
- [26] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the 8<sup>th</sup> Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 394–410, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, ACM SIGPLAN Notices 28(10) Oct. 1993.
- [27] M. Huchard and H. Leblanc. Computing interfaces in Java. In *Proceedings of the 15<sup>th</sup> IEEE Conference on Automated Software Engineering*, pages 317–320, Grenoble, France, 2000. ASE '00, IEEE Computer Society Press.

- [28] IWPC '01. *9<sup>th</sup> International Workshop on Program Comprehension*, Toronto, Canada, May 2001. IEEE Computer Society Press.
- [29] Javadoc homepage. <http://java.sun.com/j2se/javadoc>.
- [30] Java documentation enhancer homepage. <http://www.alphaworks.ibm.com/tech/docenhancer>.
- [31] Jchempaint project homepage. <http://jchempaint.sourceforge.net>.
- [32] Jmol project homepage. <http://jmol.sourceforge.net>.
- [33] W. Korman and W. Griswold. Elbereth: Tool support for refactoring Java programs. Technical Report CS98-590, Dept. of Comp. Sci. & Eng., UCSD, July 1998.
- [34] G. E. Krasner and S. T. Pope. A cookbook for using the model view controller user interface paradigm in SMALLTALK-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug.-Sept. 1988.
- [35] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the 8<sup>th</sup> International Workshop on Program Comprehension*, Limerick, Ireland, June 2000. IWPC '00, IEEE Computer Society Press.
- [36] S. O. Kuznetsov and S. A. Obedkov. Comparing performance of algorithms for generating concept lattices. In *Proceedings of the 9<sup>th</sup> IEEE International Conference on Conceptual Structures*, pages 35–47, Stanford University, California, USA, July 2001. ICCS '01.
- [37] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of the 16<sup>th</sup> Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 300–311, Tampa Bay, Florida, Oct. 14–18 2001. OOPSLA'01, ACM SIGPLAN Notices 36(10) Oct. 2001.
- [38] C. Lindig. Concept-based component retrieval. In *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, Montreal, Aug. 1995.
- [39] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19<sup>th</sup> International Conference on Software Engineering*, pages 349–359. ICSE '97, IEEE Computer Society Press, 1997.
- [40] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, Dec. 1976.
- [41] B. Meyer. *EIFFEL: The Language*. Object-Oriented Series. Prentice-Hall, 1992.
- [42] B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice-Hall Object-Oriented. Prentice-Hall, 1994.
- [43] L. Nourine and O. Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71(5–6):199–204, Sept. 1999.
- [44] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.
- [45] Seneca project homepage. <http://seneca.sourceforge.net>.

- [46] C. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 1948.
- [47] Y. Shen and Y. Park. Concept-based retrieval of classes using access behavior of methods. In *Proc. of the Int. Conf. on Inf. Reuse and Integration*, pages 109–114, 1999.
- [48] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 170–179, Bari, Italy, Oct. 1997. ICSM '97, IEEE Computer Society Press.
- [49] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. on Soft. Eng. & Metho*, 5(2):146–189, 1996.
- [50] G. Snelting. Concept analysis – a new framework for program understanding. In *ACM SIG-PLAN/SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–10. ACM Press, 1998.
- [51] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. Prog. Lang. Syst.*, 22(3):540–582, 2000.
- [52] C. Steinbeck, D. Gezelter, B. A. Smith, E. Luttmann, and E. L. Willighagen. The chemistry development kit (CDK): A Java library for structural chemo- and bioinformatics. *Journal of Chemical Information and Computer Sciences*, 2003.
- [53] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. on Sys., Man, and Cybernetics*, 11:109–125, 1981.
- [54] P. Tonella. Concept analysis for module restructuring. *IEEE Trans. Softw. Eng.*, 27(4):351–363, April 2001.
- [55] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 230–240, Oxford, England, Sept. 1999. ICSM '99, IEEE Computer Society Press.
- [56] Unified modeling language homepage. <http://www.omg.org/uml>.
- [57] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21<sup>st</sup> International Conference on Software Engineering*, pages 246–255. ICSE '99, IEEE Computer Society Press, 1999.
- [58] R. Waters, S. Rugaber, and G. D. Abowd. Architectural element matching using concept analysis. In *Proceedings of the 14<sup>th</sup> IEEE Conference on Automated Software Engineering*, pages 291–294, Cocoa Beach, Florida, USA, Oct. 1999. ASE '99, IEEE Computer Society Press.
- [59] D. A. Watt. *Programming Language: Concepts and Paradigms*. Prentice-Hall, 1990.
- [60] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.