Revealing Class Structure with Concept Lattices[*]

Uri Dekel[†]
ISRI, School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
udekel@cs.cmu.edu

Yossi Gil
Department of Computer Science
Technion – Israel institute of Technology
Haifa, Israel 32000
yogi@cs.technion.ac.il

Abstract

This paper promotes the use of a mathematical concept lattice *based upon the binary relation of accesses between methods and fields as a novel visualization of individual* JAVA *classes. We demonstrate in a detailed real-life case study that such a lattice is valuable for reverse-engineering purposes, in that it helps reason about the interface and structure of the class and find errors in the absence of source code. Our technique can also serve as a heuristic for automatic feature categorization, enabling it to assist efforts of re-documentation.*

1. Introduction

Belady and Lehman's [3] seminal *laws of program evolution dynamics* state that code repairs tend to destroy the structure of a software system, and increase its level of entropy (or disorder). This paper deals with the problem of understanding, analyzing, and even restoring order in large object-oriented (OO) classes whose entropy increased with time due to what is called *horizontal evolution* [27].

The code listings of large classes can span dozens of pages, and although many development environments include class browsing tools, most follow the style of offering a simple alphabetical list of the features of the class. The question which drives our curiosity here is: *Can the cohesive OO nature of a class be used to present its features in a more meaningful order and thus to systematically reveal its structure?*

Our answer is based on applying, for the first time, the technique of *formal concept analysis* (FCA) to the task of studying individual OO classes. FCA, germinated by Birkhoff [5] and considerably enriched by Ganter and Wille [14, 36], is a mathematical technique for clustering abstract entities, commonly called *objects*[1], that share common *attributes* into *formal concepts* organized in a *concept lattice*. This technique found many different applications in software engineering, such as *configuration management* [31], debugging [1], searching in software libraries [13, 23, 29], and construction of class hierarchies [16, 32].

A very prominent such application is in studying legacy, non-OO code, usually with the purpose of finding modules, and even organizing these in a hierarchical, OO-structure [2, 21, 24, 30, 35]. In such applications, objects are often the global variables of the program, while the attributes are procedures or subroutines. A formal concept is then a maximal set of variables and a maximal set of procedures such that *all variables are used by all procedures and all procedures use all variables*. Formal concepts or clusters of concepts serve as candidates for modules or classes, while the partial order relation, depicted in the lattice, makes candidates for a containment relationship between modules or module abstraction levels.

Thus, our research makes the next obvious step: apply FCA in a similar fashion to OO code, where fields take the role of global variables and methods that of procedures or programs. In doing so we can reveal the structure and improve our understanding of classes.

We describe the "context of a class" abstraction, and show its usefulness for effective class analysis. In essence, this abstraction is a tabular representation of the binary relation "method accesses field" of the class. We further augment the chest of tools available to reverse engineers with the "class sparse lattice" abstraction, a visual and topological encoding of the context abstraction. We argue that the sparse lattice provides an even more effective means of examining the class context, since groups of methods are clustered together, and edges indicate the relations between the groups.

[*]Research supported in part by the Bar-Nir Bergreen software technology center for excellence.

[†]Research was carried out while the author was a graduate student at the Technion – Israel Institute of Technology

[1]Not to be confused with the objects of object oriented programming

The rest of this paper describes both abstractions in depth, and explains why we expect them to be useful as means to obtaining a general understanding of a class, prior to more elaborate reverse-engineering, re-engineering, or code-inspection efforts. Our theoretical claims are supported, in part, by data obtained from an ensemble of circa 6,000 JAVA classes, as well as from a detailed case study. A preliminary user study [11] provides some support for these claims.

We also suggest a methodology for examining the *interface* and *implementation* of an unfamiliar large class. Its benefits to class customers are in shortening the learning curve; to developers in the ability to find inconsistencies, missing or superfluous operations; to documenters in assisting a *feature categorization* [26, pp.103–108].

This methodology was applied in a number of case studies [10], one of which serves as our main running example here. An evidence to the efficacy of the methodology is that with no background and with minimal effort, we revealed problems which were confirmed as new errors by the developers and were fixed in subsequent versions. While automatic tools may reveal some of the more localized errors, our approach assists in discovering delocalized problems which are more difficult to find and require an understanding of the class and its interface as a whole.

An extension of the methodology [9, 10] utilizes the lattice for selecting an efficient reading order of the source code, if available. We also describe the *embedded call graph*, an amalgam of sparse lattices and call graphs, which has the potential of combining the two visual methods to obtain new insights about the class.

Outline Section 2 is a concise overview of FCA, demonstrating its application for reverse-engineering a small JAVA class. The *context abstraction* and *sparse lattice abstraction* are presented in Section 3. The first stage of the methodology, which involves studying the interface and obtaining an abstract understanding of the class, is described in Section 4. Section 5 is dedicated to Stage II, in which we zoom-in into details of the implementation, all without dealing with the source code. The *embedded call graph* is described in Section 6. Finally, Section 7 concludes and outlines directions for future research.

2. Concept Analysis

This section reviews the theory of FCA and demonstrates how it can be used in studying `Pnt3D`, a simple JAVA class.

FCA starts with a *context* which is a triple $\langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$, where $\mathbf{O}$ and $\mathbf{A}$ are sets and $\mathbf{R} \subseteq \mathbf{O} \times \mathbf{A}$. We say that $\mathbf{R}$ is a *binary relation* between the *set of objects* $\mathbf{O}$ and the *set of attributes* $\mathbf{A}$. Table 1 depicts such a relation, where the set of objects consists of the 4 fields of `Pnt3D`, and the set of attributes consists of its 12 methods. Check marks denote that a field is accessed, directly or indirectly, by a method.

		attributes											
		`getX`	`setX`	`getY`	`setY`	`setXY`	`getColor`	`setColor`	`Pnt3D`	`getZ`	`setZ`	`setXYZ`	`draw`
objects	`x`	✓	✓			✓			✓			✓	✓
	`y`			✓	✓	✓			✓			✓	✓
	`z`								✓	✓	✓	✓	✓
	`color`						✓	✓	✓				✓

Table 1. Context of the *Pnt3D* class

The fact that the relation in Table 1 can be generated automatically from the *compiled* class file, makes the technique useful for reverse engineering.

Every subset of the objects, $O \subseteq \mathbf{O}$, has a corresponding subset of *common attributes*, denoted $\overline{O}$. An attribute $a \in \mathbf{A}$ is in $\overline{O}$ iff every object in $\overline{O}$ has a. Similarly, every subset of attributes, $A \subseteq \mathbf{A}$, has a corresponding set of *common objects*, $\overline{A}$, such that an object $o \in \mathbf{O}$ is in $\overline{A}$ iff it has every attribute in A.

A pair $\langle O, A \rangle$ such that $O = \overline{A}$ and $\overline{O} = A$ is called a *(formal) concept*. In the context of `Pnt3D`, one such concept is formed by the set of three fields $\{$`x`, `y`, `z`$\}$, which are all accessed by the three methods $\{$`Pnt3D`, `draw`, `setXYZ`$\}$.

A concept $c_1 = \langle O_1, A_1 \rangle$ is a *subconcept* of (or *dominated* by) concept $c_2 = \langle O_2, A_2 \rangle$, denoted $c_1 \leq c_2$, if $O_1 \subseteq O_2$ (or, equivalently $A_1 \supseteq A_2$). If there is no third concept c_3 such that $c_1 < c_3$ and $c_3 < c_2$ then c_2 dominates c_1 *directly*. Set $P(c)$ (resp. $C(c)$) is the set of concepts which directly dominate (are dominated by) c.

The partial order between concepts can be depicted as a *Hassé diagram* called a *concept lattice*. The concept lattice of class `Pnt3D` is depicted in Figure 1.

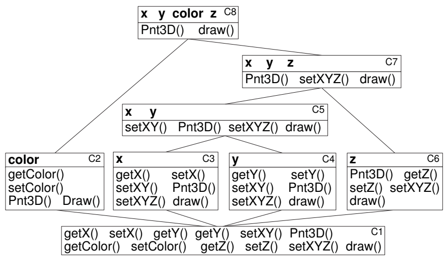
x y color z C8
Pnt3D() draw()
x y z C7
Pnt3D() setXYZ() draw()
x y C5
setXY() Pnt3D() setXYZ() draw()
color C2
getColor() setColor() Pnt3D() Draw()
x C3
getX() setX() setXY() Pnt3D() setXYZ() draw()
y C4
getY() setY() setXY() Pnt3D() setXYZ() draw()
z C6
Pnt3D() getZ() setZ() setXYZ() draw()
getX() setX() getY() getY() setXY() Pnt3D() C1
getColor() setColor() getZ() setZ() setXYZ() draw()

Figure 1. Concept lattice of the *Pnt3D* class.

Wille's *fundamental theorem on concept lattices* [36] states that every concept lattice is a *complete lattice*: The unique infimum of concepts $c_1 = \langle O_1, A_1 \rangle$ and $c_2 = \langle O_2, A_2 \rangle$ is the concept $\langle O_1 \cap O_2, A_1 \cup A_2 \rangle$, while their unique supremum is the concept $\langle O_1 \cup O_2, A_1 \cap A_2\rangle$. It follows that every lattice has a unique *top concept* (C_8 in Figure 1) and a unique *bottom concept* (C_1).

Let $n = |\mathbf{O}|$ be the number of fields of a class, and $m = |\mathbf{A}|$ be the number of its methods. Then, ℓ, the number of different concepts, might be exponential in n and m (precisely $\ell \leq 2^{\min(n,m)}$). The fact that $\ell = 8$ in `Pnt3D`, even though in this class $2^{\min(n,m)} = 16$, indicates that its fields tend to be used together.

Much redundant information is depicted in Figure 1. The *sparse lattice* (Figure 2) is a more compact representation in which fields and methods are listed only in the concept which *introduces* them; a field (a method) is introduced in the unique lowest (highest) concept in which it appears.

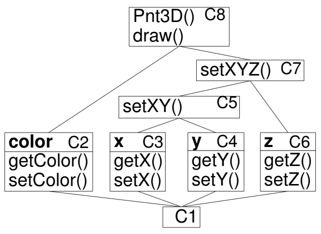

Figure 2. Sparse lattice of the *Pnt3D* class.

The sparse lattice partitions the set of methods and fields into disjoint subsets (some of which may be empty), each containing methods which use exactly the same fields, and hence likely to be related. The structure thus imposed on the method-set makes it much easier to study. All the fields used by a certain method can be collected by traversing the concepts which are dominated by the concept of this method. Conversely, all the methods which use a certain field are collected from this field's concept, and from all the concepts which dominate it.

The uncluttered representation of Figure 2 highlights its asymmetric structure. A moment's pondering reveals that the coordinates are not symmetric. The reason is probably that `Pnt3D` inherits from a class which represents a two dimensional point. One can also surmise from Figure 2 that `Pnt3D` has two main components: coordinates and color.

3. The Context- and Sparse Lattice- Abstractions, and Structured Class Exploration

The implicit assumption in the classical application of FCA to the modularization of legacy non-OO code is *variable-access module cohesion*, whose strongest version is that all the functions of a module should use all of its variables. The natural question is whether OO classes obey a similar *field-access class cohesion* assumption. In a data-set of $5,846$ non-trivial non-internal classes [10], we found that 95.5% of all classes are not strongly cohesive.

We believe, based on our own manual code inspection, that the lack of cohesion in many of these classes reveals their internal structure and indicates imperfections, inconsistencies, asymmetries and even errors of design and implementation. This claim is strengthened by the famous LCOM (lack of cohesion in methods) metric and its variants [8, 17], all expressing the belief that there should be a considerable overlap between the sets of fields used by each method.

In its most general form, our first research claim can be stated as follows:

Claim I *(The Class Context Abstraction).* The class context is a powerful means for understanding the functionality and the implementation of a class.

Large nontrivial classes, whose interface may constitute hundreds of features and whose implementation may involve dozens of fields, are the primary candidates to enjoy the context's summarizing and abstracting representation. The size of these large classes may exceed that of some modules in legacy systems, and might lead them to suffer from the same disorder imposed by continuous maintenance.

Classes of such scale are quite abundant: In our data-set, as many as a quarter of all **`public`** methods were found in classes with 100 methods or more. The *shopping list approach* [26, pp.80–83] encourages the programmer to develop large classes; it is particularly easy to do so thanks to inheritance. The laws of program evolution dynamics [3] lead us to believe that such classes tend to increase in size and consequently in complexity just like large modules in legacy software.

Instead of examining the voluminous listing of a large class, the user may choose a shorter representation which, at the cost of omitting some of the details, gives a better global grasp of the class structure. Our first claim is that the context defined by a large class is indeed an abstraction; i.e., it offers a useful global perspective of the class. Our user study [11] suggests that programmers equipped with the context abstraction become more effective in detecting certain defects in a large class.

There is also a theoretical rationale supporting Claim I: We argue that *the structure of class instances, as implied by fields, is fundamental to understanding the class.* For example, consider a class representing a process in an operating systems kernel, including data structures such as page tables, process ids, register files, lists of open files, etc. Then, the set of fields used by a synchronization operation will reveal much of the fundamentals of the operating system design.

We argue further that *the set of fields constituting the structure of a class is less volatile than the set of services* *it provides*. Consider the famous example of the alternative implementations of a complex number class using the cartesian or the polar representation. Switching between these alternatives is tantamount to rewriting the entire class, and is therefore less likely to happen than changing the services to the class. Similarly, almost every method in a class representing a rectangle would have to be rewritten when switching from a two-corners representation to a location-and-dimensions representation. We also believe that for a fixed representation, it is often the case that *all possible implementations of the same service will use the same set of fields*.

The class context can be represented in a variety of ways: as a list of facts, a table, a bipartite graph, etc. In the experimental validation of our first claim, we used a tabular representation, similar to Table 1. We argue further, that the *sparse lattice* (see e.g., Figure 2) artifact of FCA enables an even more effective examination of the context.

Claim II *(The Sparse Lattice Abstraction).* The sparse lattice is an effective means for examining the class context abstraction.

The sparse lattice and the tabular context represent the same data; each representation can be generated from the other. By definition, the sparse lattice summarizes the context abstraction in that each field and each method occurs precisely once, while similarities between rows or columns are captured by the organization of the methods and fields into concepts; identical rows and columns are effectively compressed by the concept lattice. Further, the sparse lattice organizes these formal concepts in a compact *hierarchical structure*.

On the other hand, the sparse lattice is not always shorter than the context. If the context is such that each method does not access one unique field, but all the others, then the number of concepts is exponential, although many of them are empty.

We find, however, that the tendency of fields to be used together by methods, as in the `Pnt3D` example, is a *sweeping phenomenon*: Indeed, in 99.5% of classes in our dataset, the number of concepts is linear ($n < m + f$). Moreover, in 77.4% of classes, $n < m$, i.e., in considering concepts we need to examine fewer pieces of information than in considering isolated methods. In other words, the number of concepts (including empty ones) in the sparse lattice is usually smaller than the number of rows in the context.

The second claim not only says that the sparse lattice is shorter than the context (so is a “zipped” representation), but that this lattice is effective in discovering the *internal structure* of large classes. It can help identify layers in their implementation, and may be useful for tasks such as *feature categorization* [26, pp.103–108] and documentation (since related features are placed together and organized in a hierarchy), *code inspection* [15], *requirement tracing* [4] and otherwise *re-engineering* the class.

Furthermore, we claim that one can use standard techniques of FCA in the analysis of the sparse lattice. A case in point are *horizontal-summands*, which were previously [24] used to find implicit modules in flat non-OO programs, and which might indicate that a class can be decomposed into independent units.

The case for sparse lattices is also made by our user study [11], which shows that programmers are more productive in detecting delocalized defects in the interface and implementation of a class when allowed to use this summarizing representation. The study shows that even a five-minute introduction is sufficient for users to grasp the essence of this representation, and that their productivity increases if they are offered structural aids in their work.

Claim III *(The Structured Methodology Hypothesis).* The class structure is more readily revealed by the sparse lattice abstraction when users follow a structured methodology.

It is a colossal empirical research effort to find the set of *optimal structural aids*, and then their *optimal order of application*. Our support of **Claim III** is in a demonstration of one (not-necessarily optimal) such set and order of application. Concretely, we present an FCA-based *toolbox* of views and diagrams which can be (almost) automatically generated. These tools are used in our methodology both for abstracting the class information, and for focusing on interesting details.

Our methodology is intended, in part, to improve the unstructured ad-hoc study of classes which developers make in the course of the *micro development process* [6]. In particular, it does not incur the overhead of a rigorous process, and can be invoked on a per-need basis. The tools are easy to implement, learn and use, and can be smoothly integrated into development environments.

Our running example here[2] is class `Molecule`, a large class (77 **`public`** members, over 1,500 LOC) drawn from the *Chemistry Development Kit* (CDK) [7, 33][3], an open-source library of JAVA classes for chemoinformatics and computational chemistry. The library serves as a basis for other applications, such as *JChemPaint* [19], *JMol* [20], and *Seneca* [28]. Prior to the case study selection 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.

Class `Molecule` represents an entity that should be familiar to a wide scientific audience[4]; yet it sports a large interface consisting of 77 **`public`** mem-

[2]Another detailed worked-out example, drawn from a graph-theory domain, can be found elsewhere [9, 10].

[3]We analyzed build 20020518, released in May 2002.

[4]A chemical molecule consists of atoms that are connected by bonds; it can be thought of as a graph where vertices are atoms and edges are bonds.

bers. The class **extends** `AtomContainer`, which in turn **extends** `ChemObject`. Prior to the analysis the class was *flattened* [25, p.106]; in this paper little distinction was made between members based on their origin.

Each of the next three sections describes a stage of our methodology, and demonstrates it on the `Molecule` class.

4. Stage I: Interface Analysis

In the course of presenting our methodology, we are going to show how different errors can be systematically discovered, and how an understanding of the class can be gained in the process. While some of these errors are localized and can be detected with other tools and techniques, many are delocalized and tend to evade inspectors using traditional methods. Note that we do not see error-detection as the primary goal of our methodology (automatic tools may discover many of these problems), but use it to demonstrate how our approach assists in reasoning about the class.

The first stage of our methodology is to study the class interface, where the concept lattice *partitions* the **`public`** methods into concepts and organizes them in *layers* of abstraction. Even though this stage is primarily concerned with the interface, the process is not pure, and we are sometimes forced to peek into the implementation, since, as shown by the running example, details of the implementation can sneak into the interface. Conversely, an incomplete interface definition must be elaborated by examining the implementation.

There are 7 steps or activities in this stage, which are not necessarily carried out in sequence; the first ones construct the lattice and zoom-out to obtain a general understanding, and the later ones zoom-in to investigate specific details. We now turn to describing them briefly; a more detailed discussion can be found in [10].

Step 1: ***Become familiar with the abstracted entity and the environment of the class.*** Even though the concepts and their lattice are created automatically, their interpretation can only be done by a human mental effort, to which the main clues are the names and signatures of methods. In order to make sense of these identifiers, it is essential to become familiar with the *vocabulary* and with the *human context* at which the class operates.

Step 2: ***Context selection.*** The lattice construction begins with a *selection of an appropriate class context*. For interface analysis, we start with what is called the *flat-short form* in the EIFFEL jargon [25, p.106]. The selected context consists of **`public`** methods only, regardless of their **`static`** status; methods defined in ancestors are included, unless they were overridden.[5] All the fields of the class are included in the context, regardless of their visibility and **`static`** status. The incidence relation includes read- or write-access. Note that we do not distinguish between direct and indirect access to a field. Also, as customary in the relevant literature, no alias-analysis is attempted.

[5]Methods declared in `java.lang.Object` are not included unless overridden because they are common to all JAVA classes.

Applying thus FCA to `Molecule` conveniently organizes its 75 methods and two **`public`** fields in 26 concepts.

Step 3: ***Layers-based lattice layout.*** We expect more sophisticated methods to use more fields, and hence to be located higher in the lattice. In examining many class lattices we also found that concepts at the same "layer" tend to have similar properties. For example, each of C_2, C_3, C_4, and C_6 in Figure 2 dominates the bottom concept directly. They are also similar in that each introduces a single field with an accessor and mutator for it. Figure 3 shows a partitioning of an example lattice into *layers*.

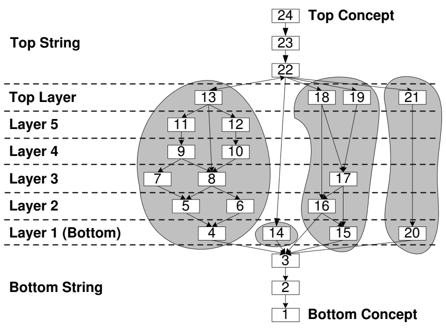

Figure 3. Layers and components in an example lattice

Formally,[6] the *bottom string* (resp. *top string*) consists of the bottom (top) concept and all concepts c such that $P(c') = \{c\}$ (resp. $C(c') = \{c\}$) for c' which is in the bottom (top) string. A concept c is in the *bottom layer*, also called *layer 1*, if c is not in the bottom string, but all its descendants are. The *top layer* is defined similarly, except that it cannot include concepts of the bottom layer. All other concepts are in *internal layers*. 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 $i - 1$.

Lattices are drawn so that concepts in the same layer appear at the same horizontal level. Figure 4 lays out the (sparse) concept lattice thus computed of class `Molecule`.[7] The figure is very clutted as it displays the full signatures of all 77 members. Nevertheless, the layout highlights the

[6]A slightly different definition of layers is employed [31, 34] to find visually pleasing layouts of lattices.

[7]Due to space restrictions, concepts C_2–C_9 appear at different heights even though they belong to the same layer, and non-`public` fields are listed although they are ignored until the second stage.

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 variables.

Step 4: *Simplify concepts' annotations.* To simplify the picture, we now try to manually *replace the list of methods in the label of each concept with a more concise, semantical description of its role*. In doing so we rely on the vocabulary and information gathered in the first step. Unknown terms and methods are prudently retained for further exploration.

In many cases, these textual descriptions can be further summarized by actually *naming the concepts*. The *responsibility legend* of Figure 5 describes our specialized notation scheme for these names, including provisions for free text and concatenation of responsibilities.[8] The figure itself depicts the *outline lattice* of `Molecule`.

Guided by the newly found concept names and the layers, an examination of Figure 5 reveals that the interface of `Molecule` can be divided into four main categories: **(i)** *Management of (nearly) the entire state*, as done in C_{23}, C_{24} and (probably) C_{25}. **(ii)** *Management of a large number of almost-independent fields in a record like fashion* (C_2–C_9). We infer that these fields are independent since no method uses them together, except for those in the first category. **(iii)** *Direct management of interdependent properties.* These features include `atomCount`, `atom`, `bond` and `bondCount`. Their interdependency is revealed by the fact that they are united in second and higher layers. **(iv)** *Other methods dealing with abstractions of ties between atoms and bonds.*

Step 5: *Horizontal decomposition.* Consider again the concept lattice of class `Pnt3D` in Figure 2. 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`. A lattice consisting of disjoint components connected by the top- and bottom- concepts is called *horizontally decomposable* (HD).

More precisely, let $\mathbf{G}$ be the undirected graph obtained from a concept lattice $\mathbf{L}$ by ignoring edge directionality and removing the top- and bottom-strings. If $\mathbf{G}$ is unconnected, then we say that $\mathbf{L}$ is *horizontally decomposable* into *components* (or *horizontal summands*), each corresponding to a connected component of $\mathbf{G}$. Singleton components are also called *trivial components*.[9] For example, the lattice in Figure 3 is HD into the four shaded components; only one of these components is trivial.

The crucial characteristic of HD 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. Functionalities are combined (if at all) only in high-level operations.

The lattice of `Molecule` (Figure 4) is HD into two components, one of which, $\{C_{15}\}$, is trivial. In examining C_{15} we see that its sole responsibility is to manage a `title` property. Even without delving into the details of the implementation, HD highlights a potential problem: `title` is not handled by the constructor which appears in C_{24} in the other component, nor by the `clone` method in C_{23}. Another probable glitch is that the field itself is **`public`** although it has an inspector and a mutator).

Further HD of the other large component yields eight trivial components (C_2–C_9), and a large non-trivial component, L, consisting of C_{10}–C_{14}, C_{16}–C_{23} and C_{26} (surrounded by a dashed line in the figure). 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 L and does not access them.

Step 6: *The abstraction lattice.* 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 levels. 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.

Consider the formal context where objects are the *concepts of the class lattice*, attributes are the *concepts in the top layer*, and an object has an attribute whenever there is a dominance relationship between the corresponding concepts. FCA will then yield the *abstraction lattice*, whose concepts are *clusters* (each with a semi-lattice structure) of concepts from the original lattice.

The abstraction lattice of Figure 5 is not very interesting (all concepts are in a single cluster). Instead we consider the abstraction lattice of component L, depicted in Figure 6. 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; cluster names were chosen manually. The abstraction lattice organizes the 14 original concepts in 8 clusters, while reducing the number of edges from 20 to 10.

The abstraction lattice heuristics groups together related operations, even if they were separated into different concepts due to differences in their (low-level) implementation.

[8] A more detailed listing and discussion of the responsibilities legend appears in [10].

[9] The literature [14] offers a slightly different definition.

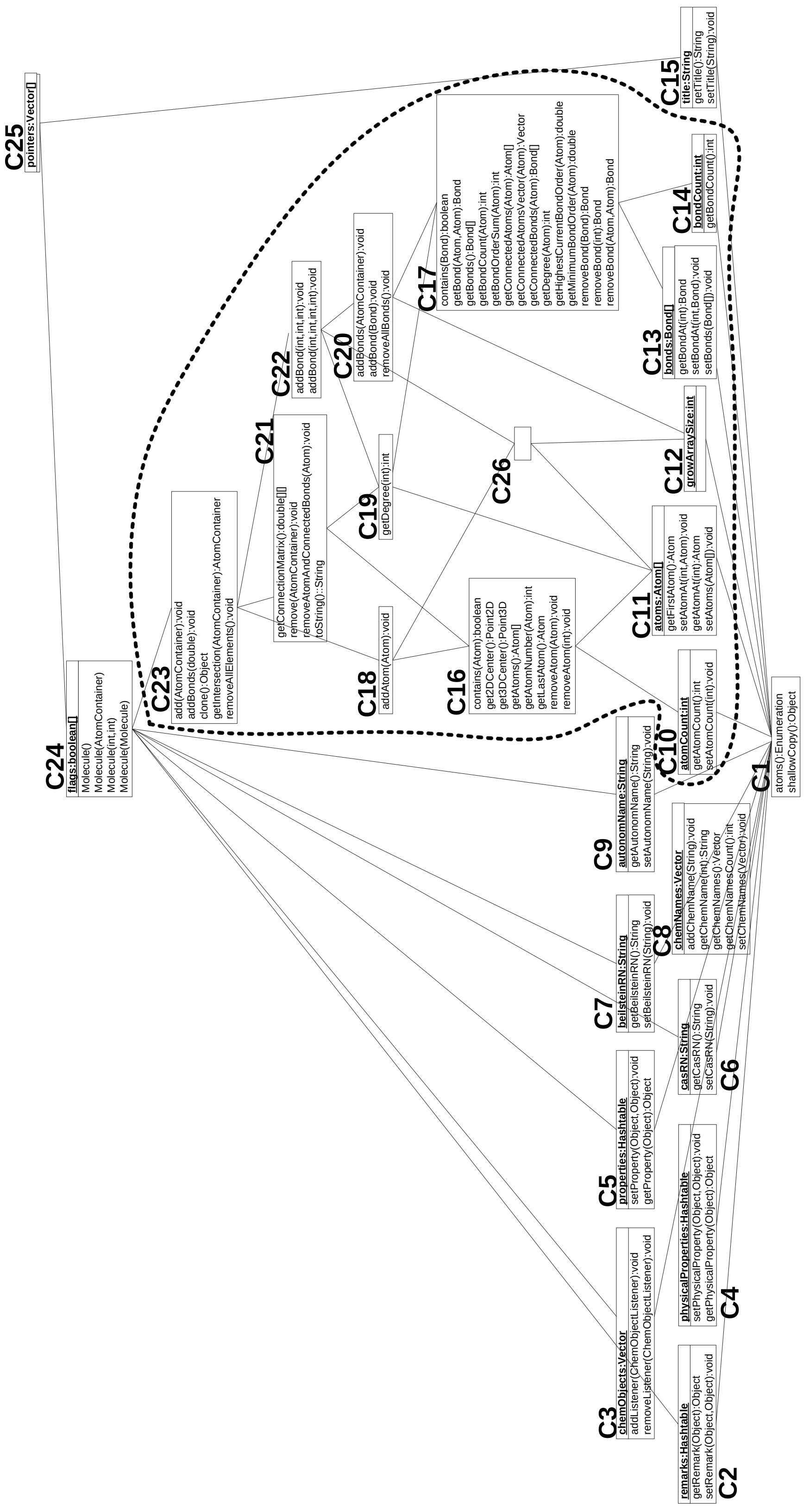
C25 pointers:Vector[]
C15 title:String getTitle():String setTitle(String):void
C14 bondCount:int getBondCount():int
C13 bonds:Bond[] getBondAt(int):Bond setBondAt(int,Bond):void setBonds(Bond[]):void
C17 contains(Bond):boolean getBond(Atom,Atom):Bond getBonds():Bond[] getBondCount(Atom):int getBondOrderSum(Atom):int getConnectedAtoms(Atom):Atom[] getConnectedAtomsVector(Atom):Vector getConnectedBonds(Atom):Bond[] getDegree(Atom):int getHighestCurrentBondOrder(Atom):double getMinimumBondOrder(Atom):double removeBond(Bond):Bond removeBond(int):Bond removeBond(Atom,Atom):Bond
C20 addBonds(AtomContainer):void addBond(Bond):void removeAllBonds():void
C22 addBond(int,int,int):void addBond(int,int,int,int):void
C21 getConnectionMatrix():double[][] remove(AtomContainer):void removeAtomAndConnectedBonds(Atom):void toString()::String
C12 growArraySize:int
C26
C19 getDegree(int):int
C23 add(AtomContainer):void addBonds(double):void clone():Object getIntersection(AtomContainer):AtomContainer removeAllElements():void
C11 atoms:Atom[] getFirstAtom():Atom setAtomAt(int,Atom):void getAtomAt(int):Atom setAtoms(Atom[]):void
C18 addAtom(Atom):void
C16 contains(Atom):boolean get2DCenter():Point2D get3DCenter():Point3D getAtoms():Atom[] getAtomNumber(Atom):int getLastAtom():Atom removeAtom(Atom):void removeAtom(int):void
C24 flags:boolean[] Molecule() Molecule(AtomContainer) Molecule(int,int) Molecule(Molecule)
C10 atomCount:int getAtomCount():int setAtomCount(int):void
C1 atoms():Enumeration shallowCopy():Object
C9 autonomName:String getAutonomName():String setAutonomName(String):void
C8 chemNames:Vector addChemName(String):void getChemName(int):String getChemNames():Vector getChemNamesCount():int setChemNames(Vector):void
C7 beilsteinRN:String getBeilsteinRN():String setBeilsteinRN(String):void
C6 casRN:String getCasRN():String setCasRN(String):void
C5 properties:Hashtable setProperty(Object,Object):void getProperty(Object):Object
C4 physicalProperties:Hashtable setPhysicalProperty(Object,Object):void getPhysicalProperty(Object):Object
C3 chemObjects:Vector addListener(ChemObjectListener):void removeListener(ChemObjectListener):void
C2 remarks:Hashtable getRemark(Object):Object setRemark(Object,Object):void

Figure 4. Concept lattice of the `Molecule` class

(Non-`public` fields, underlined, are included to conserve space, but should be ignored until Section 5. Component *L* is outlined.)

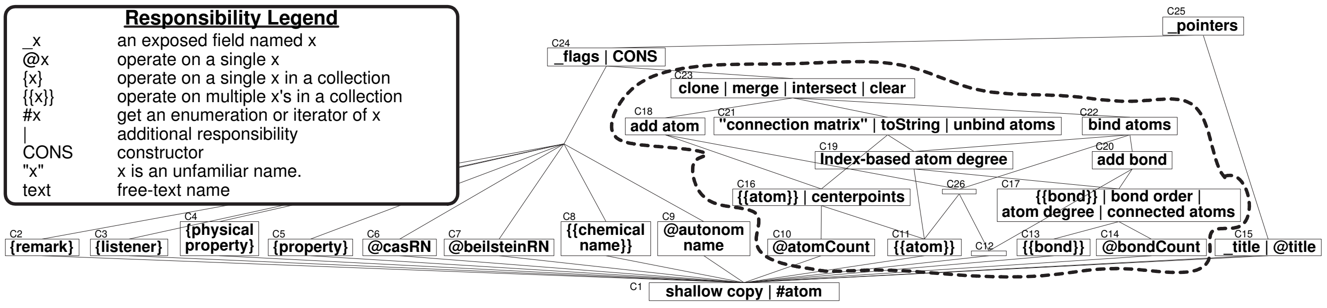
Responsibility Legend
_x an exposed field named x
@x operate on a single x
{x} operate on a single x in a collection
{{x}} operate on multiple x's in a collection
#x get an enumeration or iterator of x
| additional responsibility
CONS constructor
"x" x is an unfamiliar name.
text free-text name
C25 _pointers
C24 _flags | CONS
C23 clone | merge | intersect | clear
C18 add atom
C21 "connection matrix" | toString | unbind atoms
C22 bind atoms
C19 Index-based atom degree
C20 add bond
C16 {{atom}} | centerpoints
C26
C17 {{bond}} | bond order | atom degree | connected atoms
C2 {remark}
C3 {listener}
C4 {physical property}
C5 {property}
C6 @casRN
C7 @beilsteinRN
C8 {{chemical name}}
C9 @autonom name
C10 @atomCount
C11 {{atom}}
C12
C13 {{bond}}
C14 @bondCount
C15 _title | @title
C1 shallow copy | #atom

Figure 5. Outline lattice of `Molecule`, component L, and the responsibility legend.

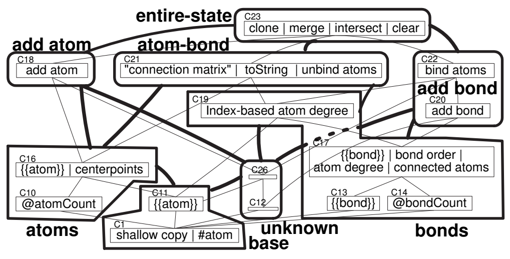
entire-state
C23 clone | merge | intersect | clear
add atom
atom-bond
C18 add atom
C21 "connection matrix" | toString | unbind atoms
C22 bind atoms
add bond
C20 add bond
C19 Index-based atom degree
C16 {{atom}} | centerpoints
C26
C17 {{bond}} | bond order | atom degree | connected atoms
C10 @atomCount
C11 {{atom}}
C12
C13 {{bond}}
C14 @bondCount
C1 shallow copy | #atom
atoms
unknown
base
bonds

Figure 6. Abstraction lattice of component L of `Molecule`

Stated differently, we 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.

Kuipers and Moonen [21] propose another clustering technique, which, unlike ours, relies on the *user* to point out related concepts, to be merged by the system. To see that automatic clustering works, consider for example concepts C_{22} (bind atoms) and C_{20} (add bond). The automatic clustering of these highlights the similarity between the services. In examining the lattice structure it is even easy to surmise that C_{22} inserts a bond between existing atoms, while C_{20} may lead to an inconsistency by binding together atoms which are in other molecules. Another example is the cluster named bonds, which reveals that the notions of "atom degree" and "bond" are highly related.

We can 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 confirmed if the code is examined) is that unknown contains utilities related to resizing the two collections.

Note that not every bit of information in the abstraction lattice is significant. For example, it is not clear why clusters base and atoms are distinct.

Step 7: ***Match services against expectations.*** It is now time to examine in detail the services supplied by the class, matching these against the expectations built in the first two steps.

The pyramid of abstractions serves as a directory of services. In searching for a functionality we may first mark all related clusters, and then zoom to concepts and methods, examining first their name, then their full signature and perhaps accompanying documentation. Partial matches against our expectations and other discoveries made along the way are dynamically added to our work list. For example, in searching for *bond management* functionality we examine clusters bonds and add bonds (atom-bond is left for future study); all the concepts in these clusters seem directly related to bond management.

The pyramidical search for functionalities and the careful examination of signatures highlights inconsistencies and other design flaws. For instance, within concept C_{17} we find that `getMinimumBondOrder` returns a **`double`**, whereas another method in the same concept, `getBond-OrderSum`, which presumably computes the sum of bond orders, returns an **`int`**. Examining methods `getHighest-CurrentBondOrder` and `getMinimumBondOrder`, occurring in the same concept, suggests that the class designers did not apply a consistent naming convention. We also find three methods: `getDegree(Atom)`, `getBondCount-(Atom)` and `getBondOrderSum(Atom)`, whose distinct responsibilities are not clear from their names.

From examining multiple concepts we discover additional problems, such as an unclear semantic distinction between the two `getBondCount` operations. We also notice hints of asymmetries and interface inconsistencies in the smaller concepts.

5. Stage II: Implementation Analysis

We now begin to delve into implementation details. At this stage, our study is carried out *without* inspecting the source code. Instead, we elaborate the class lattice by examining method signatures and adding the non-**`public`** fields which were omitted in the previous stage. By studying the fields that each method uses, and the methods that use each field, we hope to understand how the class state is realized, and how this state can be viewed and modified by the class methods. We can also utilize the *embedded call graph*, a particular representation of the *methods call graph* which can be computed from the compiled representation.

There are 7 steps in this stage, some of which 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.

Step 1: *Identify unused fields.* In the early stages of development, unused **`private`** fields are common as most methods are still stubs; such fields tend to disappear as the class nears completion. In mature classes, maintenance can result in *dead code* and *dead fields*. If such fields exist, then they will appear in the top concept, and the lattice would have a top string. In `Molecule`, the only unused field is `pointers`, but since it is **`public`**, we must check whether it is used by other clients before it can be removed. Nearby, we notice the `flags` field which is only used by the constructor.

Step 2: *Discover fields' role.* Fields are examined by their introducing concept. Each of the 9 trivial components C_2–C_9 and C_{15} represents a field with accessor methods. As the lattice shows, these fields are independent, exhibiting a record-like behavior. The role of many of them can be inferred by examining the signatures of the methods which accompany it.

We now examine component L (Outlined in Figure 4) and examine the five fields that it introduces: `atoms:Atom[]`, `atomCount:`**`int`**, `bonds:Bond[]`, `bondCount:`**`int`**, and `growArraySize:`**`int`**. The roles of the first four fields are hinted by their names: each collection is apparently maintained using an array and an associated counter field which tracks the number of occupied slots.

The role of `growArraySize` is slightly more difficult to reveal since it is used only in conjunction with other methods. Based on its name and on the fact that all the dominating concepts deal with addition and removal, we guess that `Molecule` dynamically grows the arrays of bonds and atoms, and that this field specifies the chunk size. The fact that the same field is shared for the two independent collections may be erroneous.

Step 3: *Assess the quality of field names.* When the role of each field is more or less understood, we should check that it is named properly (e.g., `mappingFromXtoY` is preferable to `hashTableOfY` because purpose is preferable to implementation). For instance, 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`.

Step 4: *Investigate fields interdependency.* The layer-structure of the lattice, and in particular non-empty second-layer concepts (or first-layer concepts with multiple fields), highlights *strong ties between fields*. In our lattice, the only such concepts are C_{16} and C_{17} in the second layer, which reveal that `atoms` and `atomCount` are closely connected, and so are `bonds` and `bondCount`. This strengthens our conviction that the count fields are used to track the number of array elements.

While some operations may be more efficient with this dual-field implementation than with a standard `Vector`, this complicates the class and introduces new risks. Namely, an important class invariant is that the number of non-empty entries in `atoms` (`bonds`) must always equal `atomCount` (`bondCount`). We make a note to validate this invariant when we examine the access patterns of individual methods.

Step 5: *Examine entire-state methods.* Some methods, often introduced in upper concepts, are intended to operate on the entire state of the class instance. Their duties often include construction, cloning, serializing, and printing. In this step we identify these methods, and check whether each of them uses all the relevant fields. Exceptions may indicate that a field is redundant and might be removed, or that there is an error in the implementation of the method.

We already learned from the outline lattice of `Molecule` (Figure 5) that the `title` and `flags` fields are not initialized or cloned due to a bug. The location of `shallowCopy` in the bottom concept indicates an obvious error in its implementation, as no fields are used. In fact, its signature indicates that it is actually a shallow-clone operation. Examining concepts C_{23} and C_{24} reveals an error in `clone` since (unlike the constructors) it only handles the fields dealing with atoms and bonds.

Step 6: *Study asymmetries.* As we saw in class `Pnt3D`, asymmetries in the class lattice can be very telling, and may indicate incomplete interfaces, inappropriate use of inheritance, and other problems. Many asymmetries are visible in component L. For example, a comparison of C_{11} and C_{13} suggests that the interface misses a `getFirstBond` method; comparison of C_{10} and C_{14} reveals another asymmetry, since there is a `setAtomCount` method, but no `setBondCount`.

Step 7: *Check method access patterns.* Now is the time to meticulously check, based on the information we collected on field names and roles, that each method uses precisely the fields that it should. By using the appropriate read-access and write-access contexts we can also check that methods make the expected kind of access.

Many flaws in `Molecule` can thus be found. The sets of atoms and bonds can be completely replaced (using `setAtoms` and `setBonds`) without updating the count. Similarly, `setAtomCount` is exposed to the user in C_{10}, allowing the invariant to be broken. We also see that the removal of atoms from the molecule does not cause incident edges to be remove.

A last step, which we shall not elaborate here, involves examining non-**`public`** methods. To do so, we must re-calculate the lattice using a context which includes all such methods.

6 The Embedded Call Graph

A concept lattice partitions the methods of the class into groups, but does not portray the interaction between them. A *methods call graph* is a powerful tool for understanding the interrelations between methods, and for making the distinction between core, auxiliary and wrapper methods. Further, as argued in the work of Lanza and Ducasse on class blueprints [22], the shape of this graph can give immediate clues on the semantical organization of the class.

To assist in the analysis of classes using our methodology, we present the *Embedded Call Graph* (ECG), a novel diagram obtained by superimposing the call graph on the class concept lattice, such that the node of each method is embedded in a block of the concept which introduces it. Figure 7 depicts the ECG of `Pnt3D`; the ECG of a larger class such as `Molecule` is much larger, and can be found elsewhere [10].

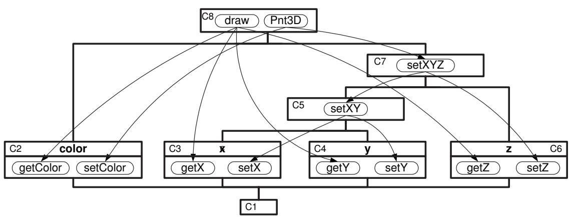

Figure 7. Embedded call graph of `Pnt3D`

The ECG can also be thought of as a semantic-driven heuristic for laying out the class call graph. The rationale is that edge crossings are minimized since, by definition, methods can only invoke methods that appear in the same concept or in dominated ones. Stated differently, recursive and mutually-recursive calls, i.e., cycles in the call graph, are always limited to a single concept. Another important property of the ECG is that if the lattice is horizontally decomposable, then the edges of different component never cross. The ECG is also useful for understanding the interrelationship between methods in the same concept or cluster.

7. Conclusions and Future Research

This research is the first to apply FCA to the analysis of individual OO classes. Our main claim is that the internal structure of a class can be revealed and reasoned about using a context lattice of the accesses relation between methods and fields. Support to this claim was provided by a theoretical rationale and a case study demonstrating its application. A preliminary user study [11] suggests that programmers can quickly learn and apply the technique.

The systematic methodology we presented supports the main claim by showing a variety of ways in which non-trivial discoveries can be made (in a semi-structured process) using this lattice: first by a mere inspection of the interface, then by delving into implementation details, and finally by a lattice-directed examination of the source. Needless to say, more practical experience is required before the methodology can be sealed and released. Other contributions include the *embedded call graph* of Section 6, and an extension of our methodology to the selection of a code inspection order [10].

We are currently exploring several research directions that build upon the foundations laid in this work. One such direction is the application of our technique to the design of classes in CASE tools. We believe that the process of adding features to a class, typically carried out by adding features to a list in a UML class hierarchy diagram, can be more effective with a lattice-based interactive editor. Methods could be associated with fields, forming concepts, and then additional methods with a related functionality could be added directly into the appropriate concept. One advantage of this approach is that it will reduce the number of cases where several methods are added for the same purpose. Another research direction focuses on the creation of a suite of class metrics based upon concept lattices and their relationship with the cohesion of the class.

In addition to theoretical work, we are currently working on the development of interactive software tools and IDE accessories that realize our technique, and in particular a plug-in for the *eclipse* [12] framework. Work on integration into documentation tools (e.g. [18]) is also carried out.

References

[1] G. Ammons, D. Mandelin, R. Bodik, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN'03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 9-11 2003. ACM Press.

[2] G. Antoniol, G. Casazza, M. D. Penta, and E. Merlo. A method to reorganize legacy systems via concept analysis. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 281–291, Toronto, Canada, May 2001. IEEE Computer Society Press.

[3] L. A. Belady and M. M. Lehman. Laws of program evolution dynamics. *IBM Syst. J.*, 15(3):225–252, 1976.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Soft-*

ware Engineering, pages 482–498. IEEE Computer Society Press, 1993.

[5] G. Birkhoff. *Lattice Theory*. Colloqulum Publications. American Mathematical Society, Providence, RI, USA, 2nd edition, 1967.

[6] G. Booch. *Object Solutions. Managing The Object-Oriented Project*. Addison-Wesley, 1996.

[7] Chemistry Development Kit (CDK) homepage. Reverse engineered with permission. `http://cdk.sourceforge.net`.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[9] U. Dekel. Applications of concept lattices to code inspection and review. In *Proceedings of the Israeli Workshop on Programming Languages and Development Environments*. IBM Haifa Research Lab, July 2002. `http://www.haifa.il.ibm.com/info/ple`.

[10] U. Dekel. Revealing Java class structure with concept lattices. Master's thesis, Department of Computer Science, Technion – Israel Institute of Techology, Haifa, Israel, February 2003.

[11] U. Dekel and Y. Gil. A user study on the use of concept lattices to detect errors in Java classes. Technical Report CS-2003-05, Department of Computer Science, Technion – Israel Institute of Techology, Haifa, Israel, June 2003.

[12] Eclipse project homepage. `http://www.eclipse.org`.

[13] B. Fischer. Specification-based browsing of software component libraries. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 74–83, Honolulu, Hawaii, USA, 1998. IEEE Computer Society Press.

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

[15] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[16] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 394–410, Washington, DC, USA, Sept. 26 - Oct. 1 1993.

[17] B. Henderson-Sellers. *Object Oriented Metrics*. Object-Oriented Series. Prentice-Hall, 1996.

[18] IBM Java documentation enhancer. `http://www.alphaworks.ibm.com/tech/docenhancer`.

[19] JChemPaint project homepage. `http://jchempaint.sourceforge.net`.

[20] JMol project homepage. `http://jmol.sourceforge.net`.

[21] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000. IEEE Computer Society Press.

[22] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 300–311, Tampa Bay, Florida, Oct. 14–18 2001.

[23] 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.

[24] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359. IEEE Computer Society Press, 1997.

[25] B. Meyer. *EIFFEL: The Language*. Object-Oriented Series. Prentice-Hall, 1992.

[26] B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice-Hall Object-Oriented. Prentice-Hall, 1994.

[27] M. Mezini. Maintaining the consistency of class libraries during their evolution. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–21, Atlanta, Georgia, Oct. 5-9 1997.

[28] Seneca project homepage. `http://seneca.sourceforge.net`.

[29] 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.

[30] 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. IEEE Computer Society Press.

[31] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. on Soft. Eng. & Metho*, 5(2):146–189, 1996.

[32] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. Prog. Lang. Syst.*, 22(3):540–582, 2000.

[33] 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.

[34] 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.

[35] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 246–255. IEEE Computer Society Press, 1999.

[36] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.