Revealing Class Structure with Concept Lattices

Uri Dekel and Yossi Gil

***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 for the study 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. The lattice can also assist in selecting an effective reading order for the source code, if available. Our results are supported by a preliminary user study.**

***Index Terms*— Reverse-Engineering, Concept Analysis, Feature Categorization, Documentation**

I. INTRODUCTION

Belady and Lehman's [1] *laws of program evolution dynamics* state that code repairs tend to destroy the structure of a software system, and increase its level of entropy. This paper deals with the problem of understanding, analyzing, and even restoring order in large object-oriented classes whose entropy increased with time due to what is called *horizontal evolution* [2].

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 classes. FCA [3] is a mathematical technique for clustering abstract entities, commonly called *objects* (not to be confused with the objects of OOP), 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 [4], software repositories [5], and the design of class hierarchies [6], [7].

A very prominent such application is in studying legacy, non-OO code, usually with the purpose of finding module candidates [8], [9]. In such applications, the global variables of the program often serve as the objects for FCA, and their attributes are the procedures or subroutines which access them. 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 manner to OO code, where fields take the role of global variables and methods that of procedures or programs. The resulting concept lattice presents an alternate view to the linear members listing, organizing them into interconnected groups which are easier to grasp. We argue that the methods within each group are likely to share a semantic similarity because they make use of the same parts of the state. In other words, we believe that field-use can serve as a heuristic for an automatic *feature categorization* [10, pp.103–108]. This organization of the methods will benefit programmers who need to quickly master the interface of a third-party class, reverse engineers who need to understand its operation, and class developers who search for defects.

In this paper we explain why the binary relation of accesses between methods and fields is a useful heuristic for categorizing features, and why presenting the interface in the form of a concept lattice is beneficial. Our theoretical claims are supported, in part, by data obtained from an ensemble of circa 6,000 JAVA classes and from a preliminary user study, as well as from a detailed case study which is presented as a running example in this paper.

The case study is demonstrated here as part of a structured three-stage methodology for studying the interface, implementation, and (when available) code of an unfamiliar large class. An evidence to the efficacy of this 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.

In addition to the methodology, we also describe the *embedded call graph*, an amalgam of concept lattices and call graphs, which has the potential of combining the two visual methods to obtain new insights about the class.

Outline Section II is a concise overview of FCA, demonstrating its application for reverse-engineering a small JAVA class. We discuss the rationale behind our approach in Section III. The first stage of our methodology involves studying the class interface and is described in Section IV. The embedded call graph is presented in Section V. Section VI is dedicated to the second stage, in which we zoom-in into the implementation details, all without dealing with the source code. When sources are available, the third stage in Section VII, shows how a sensible reading order can be selected for the purpose of carrying out an effective code inspection. Finally, Section VIII concludes and outlines directions for future research. The results of a preliminary user study are described in Appendix I.

II. CONCEPT ANALYSIS

This section reviews the theory of FCA and demonstrates how it can be used in studying `Pnt3D`, a simple JAVA class. It is important to note that the end users of our technique need not be familiar with the theory behind concept lattices.

FCA starts with a *context*, which is a triple $\langle \mathbf{O}, \mathbf{A}, \mathbf{R} \rangle$, where $\mathbf{O}$ is a set of *objects*, $\mathbf{A}$ is a set of *attributes* and $\mathbf{R} \subseteq \mathbf{O} \times \mathbf{A}$. Figure 1(a) depicts such a relation, where the set of objects consists of the four fields of `Pnt3D`, and the set of attributes consists of its twelve methods. Check marks denote that a field is accessed, directly or indirectly, by a method in at least one execution path of that method. The fact that the relation in Figure 1(a) can be generated automatically from the compiled class file, makes our technique useful for reverse engineering. Applying our technique to other programming languages would require appropriate tool support.

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 $\{\texttt{x}, \texttt{y}, \texttt{z}\}$, which are all accessed by the three methods $\{\texttt{Pnt3D}, \texttt{draw}, \texttt{setXYZ}\}$.

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*.

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(b).

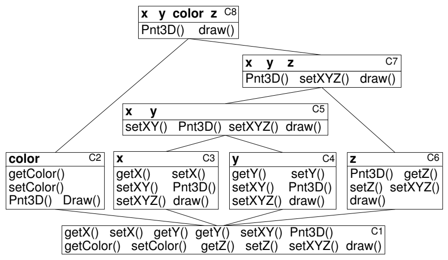
Wille's *fundamental theorem on concept lattices* [3] states that every concept lattice is a *complete lattice*: The unique infimum of concepts $c_1 = \langle O_1, A_1 \rangle$ and $c_2 = \langle O_2, A_2 \rangle$ is the concept $\langle O_1 \cap O_2, A_1 \cup A_2 \rangle$, while their unique supremum 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(b), and a unique *bottom concept* (C_1).

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

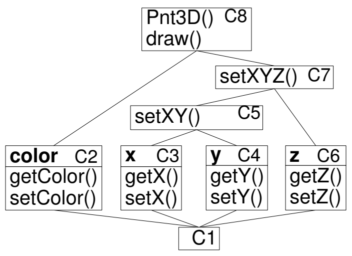
The sparse lattice partitions the set of methods and fields into disjoint subsets (some of which may be empty), each containing methods which use the same exact fields and hence likely to be related. All the fields used by a certain method can be collected by traversing the concept of this method and all the concepts which it dominates. Conversely, all the methods which use a certain field are collected from

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

(a) Context


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()

(b) Concept lattice



(c) Sparse lattice

Fig. 1. Context and lattices of the `Pnt3D` class

this field's concept, and from all the concepts which dominate it.

The structure thus imposed on the method-set makes it much easier to study. For example, the uncluttered representation of Figure 1(c) 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 1(c) that `Pnt3D` has two main components: coordinates and color.

Most real-world classes are found deep inside a class hierarchy. However, because our technique is intended to help the study of a single class in isolation, we believe that the exact details of the hierarchy are often irrelevant: It is important to understand what features the class offers and how they work, rather than what their origins are. For this reason, we *flatten* the class, aggregating features from the entire inheritance chain.

III. Rationale and research claims

Our work strives to assist developers and reverse-engineers in the study of unfamiliar classes whose interface may constitute dozens or hundreds of methods and whose implementation may involve many fields.

Classes of such scale are quite abundant: In our data-set of $5,846$ flattened classes [11], as many as a quarter of all `public` methods were found in classes with 100 methods or more. The *shopping list approach* [10, 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 [1] lead us to believe that such classes tend to increase in size and consequently in complexity just like large modules in legacy software.

A. The Class context

Instead of examining the voluminous listing of a large class, the user may choose a shorter representation which, at the cost of omitting some details, yields a better global grasp of the class structure. Thus, our first research claim is that the binary relation of accesses between methods and fields, as presented by the class context, is indeed an abstraction that provides powerful means for understanding the functionality and the implementation of a class.

Many classic applications of FCA to the modularization of non-OO code implicitly assumed *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 classes obey a similar *field-access class cohesion* assumption. The famous LCOM (Lack of Cohesion of Methods) metric [12] and its variants favored high cohesion in classes, and suggested that classes with low cohesion may require refactoring. Nevertheless, few nontrivial classes are fully cohesive. In our data-set, for example, less than 5% of classes had that property. We believe, based on our own manual code inspections, that the lack of cohesion in many of these classes reveals their internal structure. In some cases, it might also indicate imperfections, inconsistencies, asymmetries and even errors of design and implementation.

We argue that 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.

Our limited experimental validation, presented in Appendix I, supports our first claim, and shows that a context table is indeed a useful tool in its own right. However, the class context can be represented in a variety of ways, including tables and concept lattices. Our second claim, supported by the same experiment, is that the *sparse lattice* artifact of FCA enables an even more effective examination of the context.

B. Use of a sparse concept lattice

It is important to remember that although the sparse lattice and the tabular context represent the same data, the lattice often provides a more compact representation. Whereas a table can have many empty cells and identical rows or columns, each field or method in the sparse lattice is listed exactly once. Similarities or equivalencies between rows and columns are captured by the grouping of the methods and fields into concepts. The question is whether this compaction is outweighed by a large number of empty concepts.

Let $\mathbf{F}$ and $\mathbf{M}$ be the sets of fields and methods of a flattened class, respectively. The maximal number of concepts for its lattice is the size of the smaller power set of the two: $2^{\min(|\mathbf{F}|,|\mathbf{M}|)}$. In theory, a lattice can approach these exponential bounds, for example with a context where every method accesses all the fields save one, and every field is not accessed by at least one method. On the other hand, we saw the `Pnt3D` class, which had only 8 concepts despite a theoretical bound of 16. This class is not completely chaotic, and only certain combinations of fields are used together.

By constructing lattices for our data set, we found that the tendency of fields to be used together by methods, as in `Pnt3D`, is a sweeping phenomenon: Indeed, in 99.5% of classes in our data-set, the number of concepts is linear in the number of fields and methods. Moreover, in 77.4% of these classes, there are less concepts than methods, so that we need to examine fewer pieces of information than when considering isolated methods. Consider Figure 2 which plots the number of concepts vs. the number of methods for a subset of our data set, flattened classes from the JDK. As we can see, at least half of the classes have more than two methods per concept on average, and many of these have more than four.

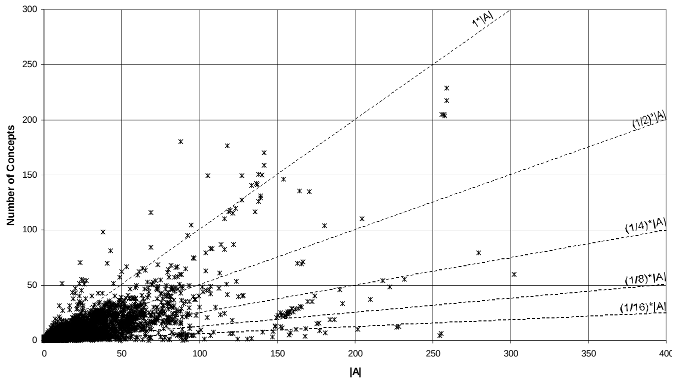


Fig. 2. Number of concepts vs. number of methods in classes from JDK 1.4.1

In summary, FCA typically places multiple methods into each concept, thus providing a more organized view of the interface. This implicit feature categorization helps those who study the class or try to document it. Furthermore, one can use standard techniques of FCA in the analysis of the lattice. For example, we shall see that *horizontal-summands* [8] may be used to suggest a decomposition of the class into independent units.

The case for sparse lattices is also made by our user study, 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 brief introduction to the interpretation of concept lattices sufficient for users to grasp the essence of this representation and discover defects in an unfamiliar class. The study also shows that providing users with guidelines on how they should approach the class and the problems they should look for often increased their performance.

C. A Methodology for studying classes

Our third claim is that the class structure is more readily revealed by the lattice when users follow a structured inspection methodology. While it is a colossal empirical research effort to find the optimal set of structural aids and then their optimal order of application, the results of our experiments suggest that even the guidelines which we provided to subjects may be beneficial. Thus, we present in this paper one possible (though not necessarily optimal) set of guidelines, in the form of a three-stage methodology which utilizes various FCA-based tools, views and diagrams. 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 perform in the course of the *micro development process* [13]. 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 is `Molecule`, a large class (77 `public` members, over 1,500 LOC) drawn from the *Chemistry Development Kit* (CDK) [14], [15][1], an open-source library of JAVA classes for chemoinformatics and computational chemistry which serve as foundation for various applications in the field. 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. A chemical molecule is essentially a group of *atoms* that are connected by *bonds*, and can be though of as a graph. Despite the simplicity of this notion, this class sports a large interface consisting of 77 `public` members. The class extends `AtomContainer`, which in turn extends `ChemObject`. Prior to the analysis the class was *flattened*, as little distinction was made between members based on their origins.

Each of the following sections describes a stage of our methodology, and demonstrates it on the `Molecule` class. Another detailed worked-out example, drawn from a graph-theory domain, can be found elsewhere [11].

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

IV. 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 elsewhere [11].

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 [16, 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.[2] 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.

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 1(c) 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*.

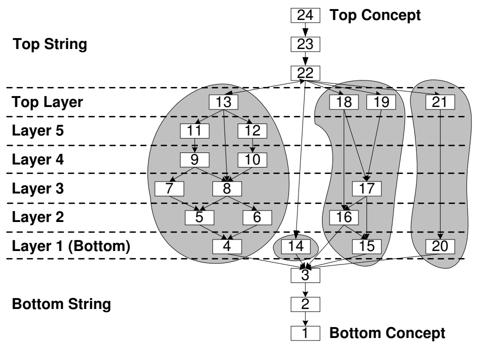


Fig. 3. Layers and components in an example lattice

Formally, the *bottom string* stretches from the bottom concept to the lowest concept that has multiple parents; the *top string* is defined similarly. Concepts which dominate only the bottom-string constitute the *bottom* (or *first*) *layer*, and those only dominated by the top string (and not in the bottom layer) constitute the *top layer*. A concept belongs to the i_{th} internal layer if 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 $i-1$.

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

Lattices are drawn so that concepts in the same layer appear at the same horizontal level. Figure 4 lays out the concept lattice thus computed of class `Molecule`.[3] The figure is very cluttered as it displays the full signatures of all 77 members. Nevertheless, 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 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, semantic 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.[4] 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.*

Note that while the current process of naming concepts is manual, we gain an understanding of the main functionality provided by the class, as methods are grouped together. Part of this process can be automated, as many methods conform to

[3]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.

[4]A more detailed listing and discussion of the responsibilities legend appears in [11].

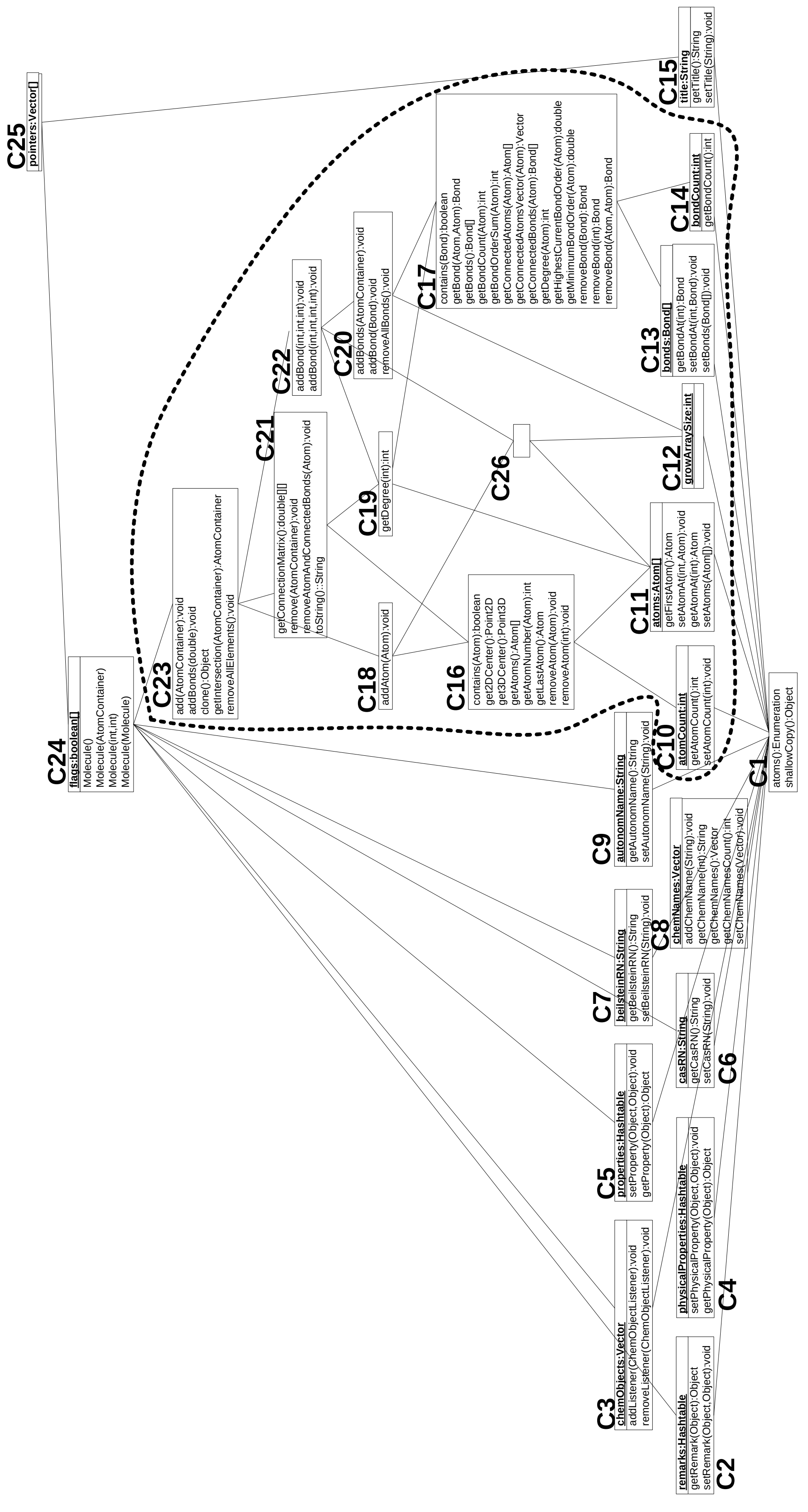
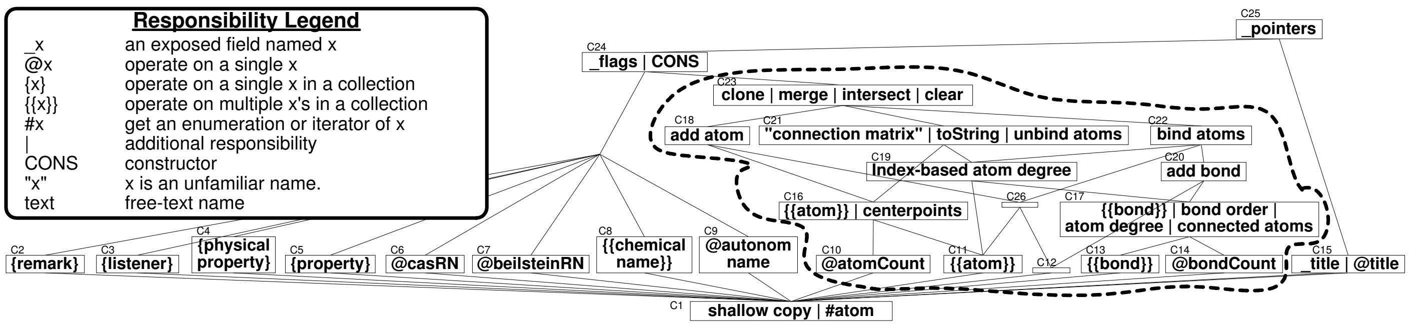


Fig. 4. Concept lattice of the `Molecule` class



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

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

micro-patterns which allow them to be automatically classified, for example as *getters*, *setters*, etc.

Step 5: *Horizontal decomposition.* Consider again the concept lattice of class `Pnt3D` in Figure 1(c). 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. A lattice consisting of disjoint components connected by the top- and bottom- concepts is called *horizontally decomposable* (HD).

More precisely, 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 component of **G**. For example, the lattice in Figure 3 is HD into the four shaded components; only one of these components is trivial (a singleton).

The crucial characteristic of HD is that methods in one component do not 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: *Construct an abstraction lattice.* It is clear that component L represents a more cohesive portion of the interface involving atoms and bonds, but the significance of each concept is lost. We further abstract the outline lattice heuristically by using methods of the top layer to group concepts at lower levels into clusters. 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 levels, then we are inclined to believe that there is a strong tie between the two sets.

We do this by constructing a context whose objects are all the concepts of the original lattice, its attributes are the concepts of the top layer, and the relation represents the dominance relationship between corresponding concepts. FCA will then yield the *abstraction lattice*, whose concepts are *clusters* of concepts from the original lattice, each with a semi-lattice lattice.

The full lattice of `Molecule` has only one top level concept, so abstracting it would be meaningless. Instead, we constructed 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.

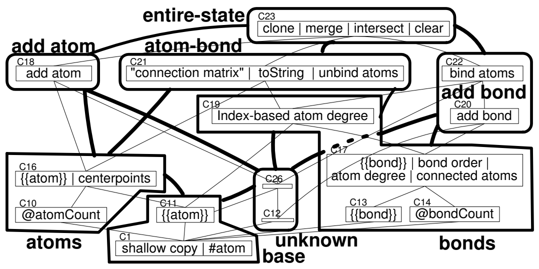


Fig. 6. Abstraction lattice of component L of `Molecule`

The abstraction lattice heuristic groups together related operations even if they were separated into different concepts due to differences in their low-level implementation. To see that automatic clustering works, consider for example concepts C_{22} (bind atoms) and C_{20} (add bond). The automatic clustering highlights the similarity between these 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. Since both the *add bond* and *add atom* clusters use the *unknown* cluster, we can make an educated guess that this cluster deals with resizing the two collections; this guess is confirmed when we later read the code. 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 set of lattices, from the original one to the abstraction lattice, constitute a pyramid of abstractions which 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, `getBondOrderSum`, which presumably computes the sum of bond orders, returns an `int`. Examining methods `getHighestCurrentBondOrder` 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.

V. The Embedded Call Graph

To fully understand the implementation of a class we must know how its methods interact, but concept lattices only partition methods into groups and do not provide this information. Method interactions are usually studied using a *methods call-graph*, which occasionally also provides clues on the semantical organization of the class [17]. Unfortunately, call graphs are usually laid out using algorithms that ignore semantics and may separate related methods.

To address these problems, we present the *Embedded Call Graph* (ECG), a novel diagram obtained by superimposing the call graph on the class concept lattice, so 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 shows the partial order between the methods within the concept. This property will later help us select a reading order for the code, as well as to determine which methods provide *core functionality* and which are merely wrappers. We can also examine a *restricted ECG*, a subgraph whose nodes are limited to specific concepts. Figure 8 depicts the restricted ECG for component L of `Molecule`.

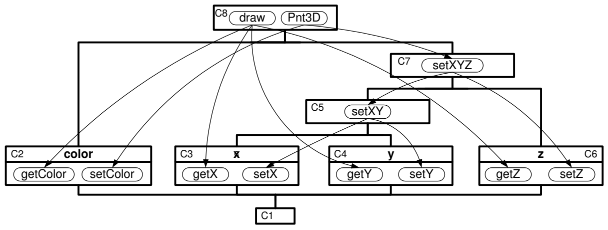


Fig. 7. Embedded call graph of `Pnt3D`

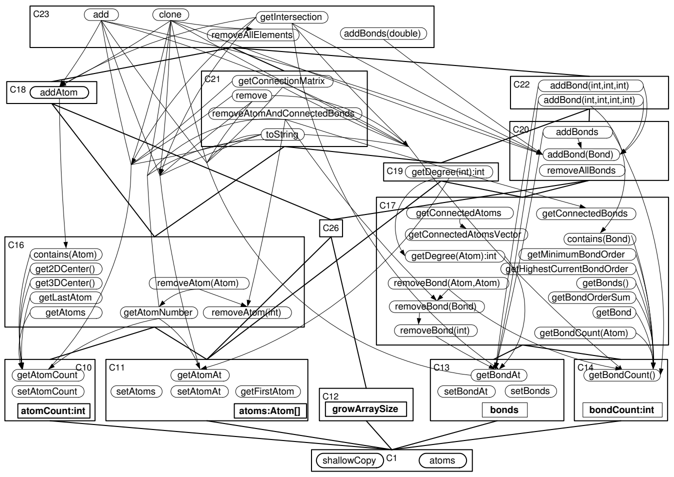


Fig. 8. Embedded call graph of component L of `Molecule`

The ECG can also be thought of as a semantic-driven heuristic for laying out the 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. Indeed, despite its cluttered look the ECG of Figure 8 has relatively few edge crossings, while preserving meaningful organization of the methods. Another important property of the ECG is that if the lattice is horizontally decomposable, then the edges of different component never cross.

VI. 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.

There are 7 steps in this stage, some of which can be thought of as check-list items of inspection. Again, 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` 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 recalculate the lattice using a context which includes all such methods. We should also determine which methods provide core functionality and which are wrappers by studying the ECG as well as method metrics [18].

VII. Stage III: Lattice-directed Code Inspection

Dunsmore, Roper, and Wood [19] argue that inheritance, dynamic binding, and small methods exaggerate delocalization effects whereby making the inspection of OO code more difficult than that of procedural code. Their experiments show that planned inspection of OO code can be more effective than an ad-hoc visit. In this section we propose a planned and sensible order for inspecting the methods of a class in isolation. We begin by making the case for this order, which will be based on the lattice structure, and then propose some specific inspection tasks lent by the lattice-directed analysis.

The source code of an OO class may be spread across several files, and may have lost any initial organization. In planning an alternative effective order by which a class is read we hope to read related methods appear together, thus increasing the probability of detecting duplications and increase opportunities for code sharing.

Another objective is to reduce the mental load of the human inspector by offering a hierarchical organization and by minimizing the number of *forward references*. This objective is also served by following the *simple-first rule* which suggests that in the absence of an ordering between items, they should be inspected in an ascending complexity order. The rationale behind this rule is obvious: the reader has to keep in mind fewer bits of information on average throughout the reading process.

We propose a lattice-based inspection order, by which the code introduced by the methods of each concept is read together. The idea is similar to Meyer's suggestion that methods be organized in groups by responsibility (global order), and then sorted lexicographically within each group (local order) [16]. The difference is that we use concepts for the responsibility-driven global order, i.e., for automatic *feature categorization*, and method invocations for the local order.

Global order: We sort concepts first by *components*, then by *ascending layer*, and finally by *cluster*. The rationale is that methods of one component never make reference to the class members, fields or methods, of another component; similarly, methods in one layer never refer to members of any higher layer. The concepts in the same cluster are expected to be related so we read these together. To choose between components, we apply the simple-first rule.

Local order: The embedded call graph is used to determine the order in which methods of the same concept are read, so that whenever possible a method is not read before the members to which it refers. Cycles within a concept and ties between independent methods are broken by the simple-first rule. Although method simplicity may be subjective, one can use *method metrics* [18] as a reasonable approximation: simple methods tend to short, contain few branching instructions (low McCabe complexity [20]), and invoke few other methods.

While applying the first two stages of our methodology we encountered questions which could only be answered by the actual code and deferred handling them. Examples include the meanings of certain terms or the roles of specific methods and fields, as well as the reasons for various inconsistencies. In addition to answering these class-specific question, we give here examples of general inspection tasks, inspired by the concept lattice approach.

Task 1: *Find duplicate services.* If two methods provide the same functionality, they are expected to use the same set of fields regardless of implementation details, and thus to appear in the same concept. In our running example, we find that C_{17} of `Molecule` introduces methods `getDegree(Atom)` and `getBondCount(Atom)` which supply the same exact service despite the difference in names and implementations.

Task 2: *Identify code-sharing opportunities.* By a similar rationale, if two methods carry out a similar computation, then these methods are likely to be in the same or a neighboring concept. Such methods may suffer from code replication, especially if they invoke the same methods. For example, the methods `addBond(int,int,int)` and `addBond(int, int,int,int)` in C_{22} serve different purposes but have almost identical implementation, suggesting a refactoring. In some cases, we may be able to make one method a composite of the other.

Task 3: *Verify that low-level methods are not bypassed.* OO-evangelists advocate that we trust optimizers enough to introduce and use "trivial" methods even for simple operations such as field access or delegation. Such methods tend to be in low-level concepts. Neglect or a misguided temptation to optimize are probably the reasons why we found so many cases in which higher level methods *bypass* the supplied trivial methods. In this inspection task we search high-level methods for code fragments which could be replaced by (existing) low-level methods. One way to do this is to manually feed the patterns of bypassed methods in low-level concepts into a *star diagrams* [21] engine. We suggest using the ECG and examining the edges emanating from methods in higher concepts and searching for methods which do not invoke methods in the lower concepts.

VIII. 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 concept 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 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 code. Needless to say, more practical experience is required before the methodology can be sealed and released. Another independent contribution is the *embedded call graph*.

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* [22] framework. Work on integration into documentation tools (e.g. [23]) is also carried out.

APPENDIX I
PRELIMINARY USER STUDY

In an attempt to provide initial validation to our hypothesis that concept lattices can assist developers studying a class, we conducted a limited preliminary user study.[5] Our study investigated the ability of novice developers who were not previously familiar with concept lattices to rapidly utilize them to discover errors in the `Molecule` class. We chose to focus on error detection because it can be measured quantitatively in short sessions, unlike more abstract notions such as understanding. We plan a more extensive user study in the future, which will provide stronger validation without the many confounding issues which are discussed later.

[5]The study is covered in detail elsewhere [24].

A. Setup and hypotheses

Our subject pool encompassed 114 subjects who received token monetary compensation for their participation. The majority were undergraduate students with limited development experience beyond their class work; a minority had industrial development experience, or were graduate students. In addition, only few were previously familiar with the JAVA language, and the rest were only familiar with C++. We had to use this limited subject pool because of resource limitations, but we intend to use a large group of experienced JAVA developers for the larger study.

The subject pool was divided into six groups which differed in the technique they had to utilize, and in whether they received a checklist. Each subject received either the code of `Molecule` and its superclasses, a single table of all the accesses between methods and fields in the flattened class, or a concept lattice for the same table. Some subjects also received a checklist, which gave ideas on the kinds of tasks or defects they should perform or look for, based on the methodology presented in this paper. The rest were not given any instructions and were simply told to find style or implementation defects, except for code readers who were also instructed to ignore style problems within method bodies. The division of participants in each experiment group can be seen on the left side of Table I.

		Number of Subjects			Average Defect Discovery Rate											
					Single-Method			Single-Field			2-way			n-way		
	Checklist	Yes	No	Total	Yes	No	Wavg	Yes	No	Wavg	Yes	No	Wavg	Yes	No	Wavg
Group	Lattice	38	13	51	28%	19%	22%	31%	26%	27%	17%	13%	14%	18%	10%	12%
	Table	19	17	36	22%	21%	22%	47%	25%	35%	1%	10%	6%	8%	2%	5%
	Code	14	13	27	16%	14%	15%	23%	7%	15%	12%	8%	10%	3%	7%	5%

TABLE I

AVERAGE DEFECT DISCOVERY RATES IN USER STUDY

All subjects filled a background questionnaire and then a short explanation on how concept lattices are read, using the lattice of `Pnt3D` from Figure 1(c). They were then given the materials and exactly 45 minutes to study the classes and find errors.

For the purpose of the experiment, we divided the defects in the `Molecule` class into four categories: *single-method* and *single-field* errors, such as problems in the access patterns or signature of a particular method, *2-way* errors which arise from differences between two interface entities, and *n-way* errors which arise from differences between multiple entities.

We had several hypotheses. First, we expected subjects using the lattices to be more successful in discovering *2-way* problems than those in the other modalities. We expected this effect to be even stronger for *n-way* defects, because they are less likely to be noticed unless the involved methods are examined together, as occurs thanks to the way the lattice clusters methods. On the other hand, we expected the lattice to be less valuable for problems that involve a single method or field, and thus be on par with users of the table. In fact, a table might be slightly more effective for these problems simply because most people are familiar with tables. It is important to remember that the lattice and the table convey equivalent information, and differences in performance are likely to be due to cognitive differences.

Another hypothesis was that subjects who use the code will perform poorly on the kinds of defects studied here because they will be distracted by problems in the internal style of the methods, and by the difficulty in ascertaining what fields are used by each method.

B. Results and analysis

The response forms filled by the subjects allowed them to list as many errors as they wanted in free form text; we later examined these forms and marked discovered defects which matched those in our list. This manual interpretation may confound the results, but we preferred using free-text responses rather than questionnaires which might lead subjects to specific defects. For each specific defect, we calculated the percentage of subjects within the same modality who discovered the same defect. Since there are multiple defects in each category, we averaged the results to obtain an average defect detection rate within the category.

The results for all modalities and categories appear in Table I. Note that since some categories contain more defects, their results are likely to be lower in absolute numbers than of those with few defects. The results should therefore be compared within the category, but between the different modalities. For each category, we also added a column of a weighted average with respect to the use of checklist and the number of subjects in each modality.

The results of this preliminary study confirm our first hypothesis, that the concept lattice is more effective than other modalities for the detection of defects that involve multiple members. This effect was especially significant for *n-way* defects with subjects who did not receive the checklist, indicating that the lattice increases the chances of discovering these inconsistencies which are otherwise likely to go unnoticed. The similarity in results between table and lattice users for single-entity defects confirm our second hypothesis, with the lattice at a slight disadvantage.

The results for the code group, however, did not fully confirm our third hypothesis. While those reading the code performed worse than those using the tables or lattices, the effect was less significant than expected, especially for n-way defects. Upon investigating the results for specific defect types, we discovered very high variance between in the results of the code group for the defects in each category. It seems that code readers performed exceptionally well on certain errors, such as with the clearly erroneous implementation of the `clone` operation and cases where subsequent placement of related methods made error discovery almost automatic. The good performance on such defects compensated for the poor results on the rest of the errors, leading to relatively high average detection rates.

The results are ambiguous regarding the usefulness of the checklist. In most cases, they seem to significantly improve the chances of discovering relevant errors, but in two modes they actually decreased performance. Further study is necessary to determine the cause for these discrepancies. One possibility is that users following the checklist often did not reach certain items within the allotted time, skewing the results.

Because of resource limitations we had to use a small heterogenous subject pool consisting mostly of undergraduate students without significant development experience, familiarity with large classes, or or knowledge of JAVA. We also had to limit sessions to 45 minutes, which were insufficient for dealing with a class of this magnitude, as evident from the small number of participants who completed all items on the checklist. Nevertheless, the results from this preliminary user study provide initial confirmation for our hypothesis and demonstrate the potential of concept lattices in the study of classes. They also provide incentive for the planned larger-scale study on experienced developers.

References

[1] L. A. Belady and M. M. Lehman, "Laws of program evolution dynamics," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[2] 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*, Atlanta, Georgia, Oct. 5-9 1997, pp. 1–21.

[3] R. Wille, "Restructuring lattice theory: An approach based on hierarchies of concepts," in *Ordered Sets*, I. Rival, Ed. Reidel, 1982, pp. 445–470.

[4] G. Snelting, "Reengineering of configurations based on mathematical concept analysis," *ACM Trans. on Soft. Eng. & Metho*, vol. 5, no. 2, pp. 146–189, 1996.

[5] C. Lindig, "Concept-based component retrieval," in *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, Montreal, Aug. 1995, pp. 21–25.

[6] 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*, Washington, DC, USA, Sept. 26 - Oct. 1 1993, pp. 394–410.

[7] G. Snelting and F. Tip, "Understanding class hierarchies using concept analysis," *ACM Trans. on Programming Languages and Systems*, vol. 22, no. 3, pp. 540–582, 2000.

[8] M. Siff and T. Reps, "Identifying modules via concept analysis," in *Proceedings of the IEEE International Conference on Software Maintenance*. Bari, Italy: IEEE Computer Society Press, Oct. 1997, pp. 170–179.

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

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

[11] 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. [Online]. Available: http://www.uridekel.com/research/Msc/udekel_mscthesis.pdf

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

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

[14] "Chemistry Development Kit," reverse engineered with permission. [Online]. Available: http://cdk.sourceforge.net

[15] 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.

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

[17] 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*, Tampa Bay, Florida, Oct. 14–18 2001, pp. 300–311.

[18] T. Cohen and J. Y. Gil, "Self-calibration of metrics of Java methods," in *Proceedings of the 26rd International Conference on Technology of Object-Oriented Languages and Systems*. Sydney, Australia: Prentice-Hall, Nov. 20-23 2000, pp. 94–106.

[19] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, 2000, pp. 467–476.

[20] T. J. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[21] R. W. Bowdidge and W. G. Griswold, "Supporting the restructuring of data abstractions through manipulation of a program visualization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 109–157, 1998.

[22] "Eclipse project homepage." [Online]. Available: http://www.eclipse.org

[23] "IBM Java documentation enhancer." [Online]. Available: http://www.alphaworks.ibm.com/tech/docenhancer

[24] U. Dekel and Y. Gil, "A user study on the use of concept lattices to detect errors in Java classes," Department of Computer Science, Technion – Israel Institute of Techology, Haifa, Israel, Tech. Rep., November 2003. [Online]. Available: http://www.uridekel.com/research/Msc/fca_userstudy.pdf

[25] U. Dekel and Y. Gil, "Revealing class structure with concept lattices," in *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, B.C., Canada, November 2003, pp. 353–364.