

Applications of Concept Lattices to Code Inspection and Review

URI DEKEL

Department of Computer Science, Technion
Technion City, Haifa, Israel, 32000

udekel@cs.technion.ac.il

Abstract

This work investigates the application of *formal concept analysis* to the tasks of reverse-engineering and code inspection of individual JAVA classes. Our technique partitions the methods of the class according to their use of fields, and then presents them in the form of a *concept lattice* (or *Galois lattice*). We demonstrate in a case study that this lattice can help visualize and understand the structure of the class. We also show how this lattice can be used to perform version comparisons and to select an order in which methods can be read effectively.

1 Introduction

The simple technique of reading source code is used extensively in software understanding and maintenance. Reading the source code is at times the only possible way to understand an undocumented system. It is also a common means of verifying the correctness and quality of code, an activity known as *code inspection* [1].

Although code inspection is usually performed on entire systems, even the understanding of a single large class is far from being trivial. The size and complexity of certain classes can overwhelm a reader, allowing various code replications and defects to go unnoticed.

We argue that reading the source code of a class in an object oriented language is significantly more difficult than reading the source code of a module in a procedural language. Part of the reason for this difference has to do with the evolution from procedural languages, in which mutually recursive definitions are rare, to object oriented languages, in which such definitions become common.

In most procedural languages (e.g. C, PASCAL, and ML), there are few cases in which mutually recursive definitions are required. These rare cases are resolved using *forward declarations* (as with the **forward** keyword in PASCAL), or using *co-definitions* [2] (as in ML). In contrast, in the definitions of classes in object oriented languages, co-definitions are the rule rather than the exception.

This difference is the reason for one of the major changes in the the evolution from C to C++. In C, the definitions within a module are sequential, and the module is processed incrementally by the compiler. The scarcity of mutual recursions allows the compiler to process the entire module in a single pass. It also facilitates reading the source code of the module, since in-

cremental understanding is within the capabilities of the human mind. The situation is different in C++ and in all other major object oriented languages, including JAVA and EIFFEL. The definitions of a class are of the co- rather than of the sequential kind: All class members are considered to be defined at the same time.

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

Understanding a class involves two tasks. First, we must understand the functionality and structure of the entire class, and identify groups of methods with similar responsibilities. Next, we must understand the purpose and implementation of each and every method. These two tasks are in fact interdependent: As we read individual methods, we learn more about the overall responsibility and structure of the class; this knowledge helps us understand other methods better.

In this work we investigate the possibility of applying *formal concept analysis* to the problem of understanding a single JAVA class. Our solution is limited (at this stage) in that it ignores the interactions of the class under inspection with other classes.

Formal concept analysis [3, 4] is a mathematical technique which was applied to different problems in software research (e.g. [5–8]). The basic idea of our methodology is to use concept analysis to partition the methods of a class into groups called *concepts*, and present these groups in the form of a *concept lattice* (or “*Galois lattice*”). The concept lattice helps visualize the structure of the class, and can aid us in selecting an effective order for reading the methods.

Our partitioning of methods into concepts is based on their use of fields. We believe that the usage patterns of data members by function members are fundamental to understanding the functionality and the implementation of a class. We argue that the fields of a class are less volatile than its methods, and that usually all possible implementations of the same operation will use the same fields.

Outline: The rest of this paper is organized as follows. Section 2 explains concept lattices and demonstrates their use for

reverse-engineering a small class. Section 4 discusses how concept lattices can be used to select a useful order of reading methods. Section 5 demonstrates our methodology in a real-life case study and shows its effectiveness for version comparisons. Finally, our current research directions are presented in Section 6.

2 Concept Analysis

In this section, we demonstrate the use of concept analysis on a JAVA class named `Point3D`, for which we are supplied only with a binary classfile.

First, an automated program extracts from the classfile all the information about invocation of methods and accesses to fields. Using standard static analysis techniques, this information is used to calculate all the accesses to fields that each method makes, including indirect accesses.

Concept analysis is performed on a binary relation called *context*, between a set of *instances* and a set of *features*; it specifies the features that each instance owns. In the analysis of the `Point3D` class, we use its four fields as instances and its fourteen methods as features. Hence, we use a context that specifies the methods which use each field.

Fig. 1 depicts the context of `Point3D`.

		features													
		draw ()	setXYZ ()	setZ ()	getZ ()	Point3D	setColor ()	getColor ()	setXY ()	setY ()	getY ()	setX ()	getX ()	color	
instances	x	✓	✓			✓						✓	✓		
	y			✓	✓					✓				✓	
	z					✓	✓	✓					✓	✓	
	color						✓	✓	✓					✓	

Figure 1: Context of the `Point3D` class

We can now use standard concept formation algorithms [4] to create a concept lattice from the context in Fig. 1. The resulting lattice of `Point3D` is shown in Fig. 2.

The nodes of the lattice in Fig. 2 are called *concepts*. Every concept contains a (possibly empty) set of fields, and a (possibly empty) set of methods.

An important property of all the methods in a particular concept is that they all use the same combination of fields. This combination is the union of the fields which appear in that concept and in every concept that it *dominates*. We say that every concept *dominates* all of its children, and that it dominates its immediate children *directly*. For example, the methods of concept #3 use only the `x` field, and the methods of concept #4 use only the `y` field. The method `setXY` of concept #5, which dominates concepts #3 and #4, uses both `x` and `y`.

Using the lattice, we can make several deductions about the `Point3D` class:

1. Every method uses at least one field, but no field is used by all methods. We know this because the bottom concept

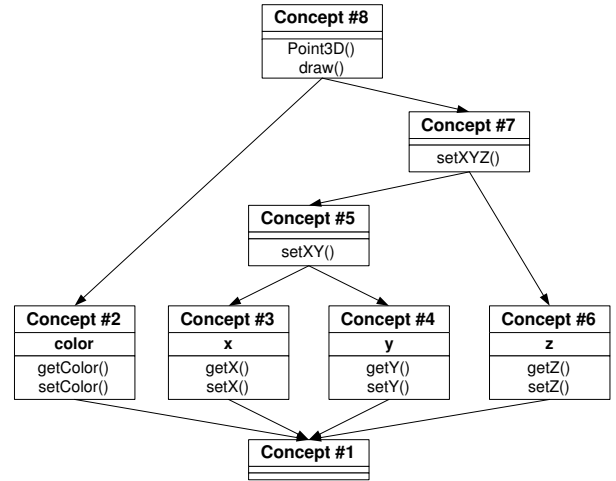


Figure 2: Concept lattice of the `Point3D` class

(concept #1) is empty.

2. There is a certain cohesion between all the fields of the class: All of them are required together to accomplish the drawing responsibility of the class. We deduce this because the top concept (concept #8) contains the nontrivial `draw` method.
3. The class has two parts. One part, consisting of concepts #3–7, is responsible for maintaining the three coordinates. The other part, consisting only of concept #2, is responsible for maintaining color information. These parts are obtained by removing the top concept and the bottom concept, resulting in two disjoint graph components. We know that some cohesion exists between the two parts because of the `draw` method in the top concept. This method indicates that both parts are required in order to implement a certain responsibility of the class. A redesign of class `Point3D` can make it into an aggregate of two classes, `Coordinate` and `Color`.
4. There is a lack of symmetry between the three coordinate components, visually evident from the existence of concept #5. This asymmetry and the lack of two complementing methods, `setYZ` and `setZX`, hints that the `Z` field might have been added to the class at a late stage of its evolution. It is also possible that the asymmetry results from inheritance, with fields `X` and `Y` defined in some superclass that represents a point in two-dimensional space.

3 Embedded Call Graph

Even though a concept lattice clusters methods that use the same set of fields, it does not provide information about the interaction between these methods. Examining the concept lattice does not reveal whether a method accesses a combination of fields directly, by accessing their values, or indirectly, by invoking methods that access them directly. By superimposing the *method call-graph* on the class concept lattice, we obtain the *embedded call-graph* (ECG) which provides a more de-

tailed visualization of the class.

A *method call-graph* is a graph in which nodes represent methods and edges represent method-inocations. This graph is a common means of visualizing the interaction between the methods of a class. For example, Lanza and Ducasse [9] use an augmented call-graph to create an outline of a class.

In the *embedded call-graph*, the methods of each concept are clustered together. Clusters are explicitly marked and are connected with edges, creating the lattice structure.

In order to demonstrate how the embedded call graph can provide important information which does not appear in the concept lattice, suppose that the `Point3D` class contains an additional method, named `setYX`, which modifies the values of the `x` and `y` fields. If method `setYX` modifies the two fields by invoking methods `setX` and `setY` directly, then the corresponding ECG is the one in Fig. 3, which shows the symmetry between methods `setYX` and `setXY`. If, on the other hand, method `setYX` modifies the fields indirectly by invoking `setXY`, then the corresponding ECG is the one in Fig. 4, which reflects that invocation.

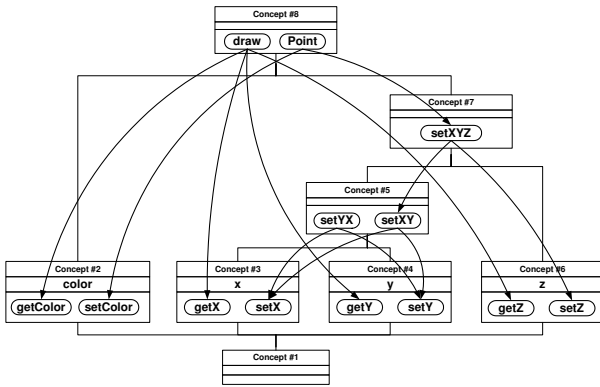


Figure 3: The embedded call-graph of class `Point3D` (Method `setYX` accesses field mutators directly)

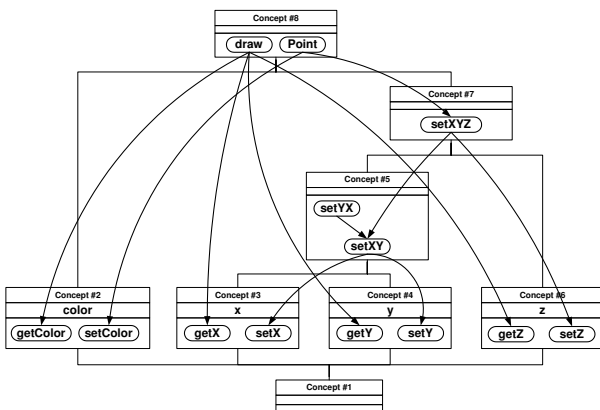


Figure 4: The embedded call-graph of class `Point3D` (Method `setYX` accesses field mutators indirectly)

We argue that a method call-graph is not practical for investigating a large class because of two problems. The first prob-

lem is crossing edges. In object oriented programming, there is a large number of method calls, and in JAVA, which does not support inlining, this number is even higher. A large number of method calls results in numerous crossing edges, making the tracing of individual calls more difficult.

To alleviate the crossing edges problem, graph layout algorithms are used to minimize crossings. Because these algorithms do not take into account the semantics and similarities between methods, they cause a second problem. Related methods may be placed far apart in the resulting graph. For example, if a class contains an inspector and a mutator for a particular field, these methods are not necessarily placed together because they do not call each other.

Unlike general graph layout algorithms, an ECG layout is based on semantics, and clusters related methods together. Also, we believe that because of its inherent properties, an ECG layout has less crossing edges than an unoptimized graph layout. We base this claim on the property that an edge leaving a method in some concept in the ECG can only reach a method in the same concept or in another concept that is dominated by the first. Therefore, the edges for method calls that occur in separate parts of the class do not cross in the ECG.

In addition, we expect that many edges will be short and connect methods in the same concept, due to the extensive use of composite methods in object oriented programming. This use is likely to be even more extensive in JAVA than in languages such as C++ because JAVA does not permit default arguments to methods. Instead of using default arguments, JAVA programmers overload a method for which default arguments are required with a new method that invokes the original method with fixed arguments.

In this section and in the previous one we demonstrated that a concept lattice and an embedded call-graph can facilitate the reverse-engineering of a class for which the source file is not available. In the next section, we discuss how the lattice and the ECG can be used in order to efficiently read the source file, if it is available.

4 Reading Order

Empirical research [10] shows that the order in which source code is read can have a significant effect on the efficiency of code review and inspection, and on the quality of their results. The primary reason for this effect is the limitations of the human mind. Consider, for example, a class with dozens of methods, in which two methods have the same functionality but different names. We will certainly discover this redundancy if we read these two methods consecutively. Our prospects are lower if we read many other methods in between.

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

Reading methods in the order of their appearance in the

source file is not practical for large classes. As a class matures and undergoes ad-hoc changes, any predetermined order which might have existed is likely to disappear. Also, the order in which members appear in the source code of a class is not necessarily meaningful: In most object oriented language, all members are assumed to be defined simultaneously. In fact, programming language mechanisms such as inheritance and inlining can actually spread related methods across several source files.

We argue that a good candidate for a reading order should introduce structure and direction into the reading process, by defining a *global order*. The methods should be separated into conceptually-related groups with an order between them. The candidate order must also define a *local order*, an ordering between the methods inside each group, without preventing the reader from making some choices on-the-fly. For example, Meyer [11] suggests that methods in EIFFEL should be separated to groups according to their responsibilities (global order), and should be sorted lexically inside each group (local order).

We suggest a structured reading order based on four guidelines:

Guideline 1 (Reading concepts) *The methods of each concept must be read consecutively.*

Because the methods of each concept are similar, reading them without interleaving with methods of other concepts should be more efficient than reading them in an interleaved order. For example, if the methods of each concept use different class libraries, then reading these methods consecutively will decrease the chances of forgetting an important library class before we are finished reading all the methods that use it.

Guideline 2 (Topological order) *Methods must be read in a topological order (bottom-up) or a reversed topological order (top-down) of the call-graph*

Following a topological or a reverse-topological order introduces a clear direction of the reading process. Note that the order between concepts never conflicts with the topological order in the call-graph: If a method calls another method, the calling method uses at least the same fields as the called method, and is therefore placed in the same concept with the called method or in a higher one. The embedded call-graph provides a valuable tool in selecting the topological order between methods in the same concept.

Guideline 3 (Layers) *The concepts of each layer should be read consecutively.*

Layers of concepts are defined by their distance from the top and bottom concepts. We argue that concepts of the same layer often contain similar methods. For example, concepts #2-4 and #6, which constitute a layer in Fig. 2, contain only *inspectors* (“get” methods), and *mutators* (“set” methods). We are currently working on identifying the kinds of methods in each layer and the nano-patterns that they follow.

Guideline 4 (Simplest methods first) *When the previous guidelines fail to create a full order between a group of methods in a certain concept, these methods should be read from the simplest to the most complex.*

We argue that while a sequence of simple methods can be efficiently processed by the human mind, encountering a complex method between the simple methods stops the flux of reading and increases overall reading time.

Defining what makes methods simple and easy to understand is subjective. Combinations of method metrics [12] can be used to estimate the simplicity of a method. Generally, simple methods are short, contain few branching instructions (indicated by a low McCabe complexity [13]), and invoke only a few methods.

5 Case Study: The Graph Class in VGJ

VGJ (“Visualizing Graphs with JAVA”) [14] is an extensible JAVA applet for drawing graphs and testing new layout algorithms. Started as a graduate students project at Auburn university and distributed under the GNU license, the software gave rise to numerous adaptations and extensions at different institutions. With the advent of FLGL, the commercial version of VGJ, support of the original tool was discontinued.

The VGJ applet has two distinct parts: The user interface, realized by the classes in the `gui` package, and the layout algorithms, realized by the classes in the `algorithm` package. These two parts interact using the classes in the `graph` package that represent graphs and graph-elements.

The `Graph` class, which represents an entire graph, is arguably the most important class in its package. In addition to being an important means of interaction between the two parts of the applet, it is also the primary class with which the developers of new algorithm components must be familiar. For this reason, we choose to apply our methodology to the review and inspection of this class.

After analyzing the original version of `Graph`, we shall demonstrate the applicability of our methodology to version comparisons by analyzing a version which was modified by students.

5.1 Original version

The `Graph` class inherits directly from `Object`. Constructors excluded, this class sports 37 public methods and 6 private ones. The prospects of a successful understanding a class of this magnitude lie with a structured inspection.

The concept lattice depicted in Fig. 5 organizes the 37 methods in ten concepts.

We begin by quickly examining each concept in the lattice from Fig. 5 and speculating on the responsibilities of its methods and fields based on their names. In the concept lattice in Fig. 6, we have annotated each concept with a short textual description of these responsibilities.

We can now examine the structure of the lattice in Fig. 6 and attempt to draw general conclusions. Later, we will study each concept in depth.

The bottom concept is empty, preventing the existence of a part of the state to which all methods relate. The top concept contains a significant number of methods which use the complete set of fields. This fact indicates a certain cohesion between all the fields, as they are all used together for implementing some responsibilities of the class.

Concepts #8 and #9 are dominated directly by the top concept. By comparing the sets of fields that their methods use, we can identify different functional parts of the class. The methods of concept #8 use `idHash`, `lastTopId` and `nodeList`. The methods of concept #9 use `edges`, `isDirected` and `nodeList`. Therefore, this class has two functional parts: One deals with a hash of nodes and the other deals with a collection of edges. Both parts share the use of a list of nodes.

Now, we can begin a thorough examination of the methods of each concept in Fig. 5. In accordance with the proposal for a reading order, we examine the concepts in an ascending order of layers, and read all the methods of each concept before proceeding to the next one. This organized approach allows us to discover important details which are likely to go unnoticed in a linear reading order. It also allows us to predict the expected implementation of some methods and then verify our predictions, which is better than searching for errors while trying to understand the implementation of the method.

Concept #1 is the bottom concept. Because it contains no fields, we can expect its methods to be **static** and serve as utilities. Indeed, this concept contains only the short method `getIndexFromNode`, which returns a public field of a `Node` parameter. This method is mistakenly not defined as **static**. Furthermore, it does not justify cluttering the interface of `Graph`. It should be either removed, relocated to the `Node` class, or declared as **private**.

Concept #2 introduces the `nodeList` field used by both parts of the class. We examine its twelve public methods according to the length of their bytecode, and discover that the nine shortest methods simply delegate to methods in the `nodeList` field. These nine methods provide iteration capabilities over the list of nodes, and should therefore be replaced by a method for retrieving the list of nodes (allowing clients to use its iteration services directly), or preferably by a method that returns a standard enumerator or iterator.

Another interesting discovery is that the two delegators, `getNodeFromIndex` and `nodeFromIndex`, actually call the same method. Since these methods are separated by hundreds of lines of code, it is unlikely that we would have noticed this redundancy by reading methods according to their order in the source code.

From reading the two non-delegating methods, `children` and `parents`, we learn that each `Node` object maintains a list of its immediate descendants, but not of its antecedents. This information is very important to the designers of new algorithm components, as the time complexity of finding the antecedents of a node is higher than that of finding descendants. In addition, we notice a stylistic flaw in both methods: Instead of using the nine delegators we have found in this concept, these methods directly invoke the original methods of the `NodeList` class.

Concept #3 introduces the field `idHash` and the delegat-

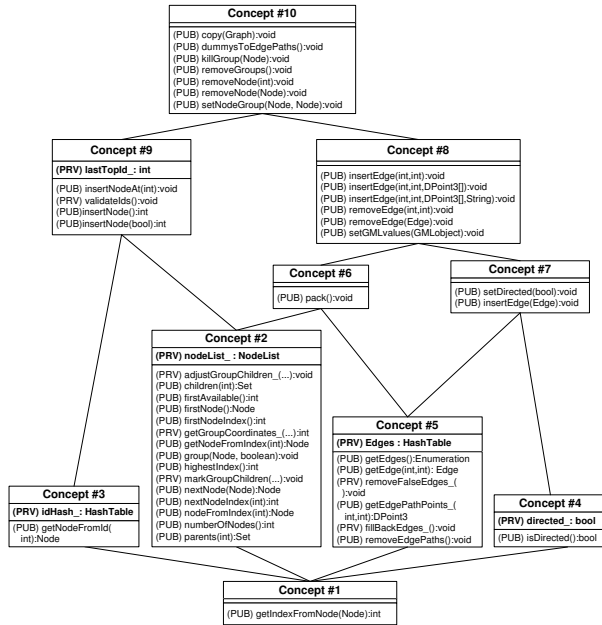


Figure 5: Concept lattice of the `Graph` class
(Concepts are arranged vertically to reflect their respective layers)

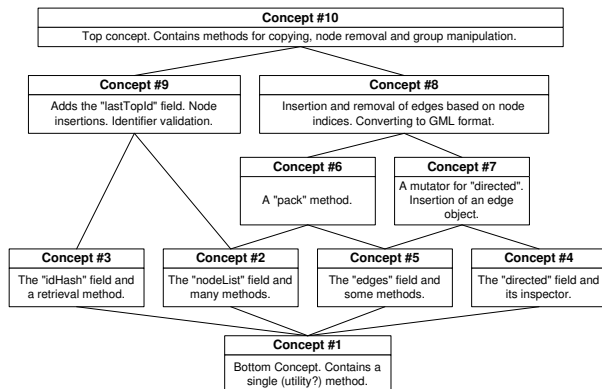


Figure 6: Simplified concept lattice of the `Graph` class

ing method `getNodeFromId`. The name of this method reveals that the field actually serves as a mapping from identifiers to nodes. Apparently, the `idHash` field was given the wrong name. The name should indicate the nature of the mapping, not the data structure used to implement it. Furthermore, the existence of a hash of nodes in addition to the list of node presents a redundancy that complicates the `Graph` class. A redesign of this class should combine these fields into a single data structure that provides both index-based and key-based retrieval of nodes.

Concept #4 introduces a boolean field whose name, `directed`, indicates that this class can represent both directed and undirected graphs. Since this concept does not contain a mutator for the field, there are two possibilities: Either the directionality of the graph cannot change after creation, or its change involves other fields. The existence of the mutator in concept #7 supports the second hypothesis.

Concept #5 introduces a field named `edges` of type `Hashtable`. We learn from the `getEdge` method that each edge is uniquely identified by the indices of the two nodes it connects. This fact implies that this class can only represent *simple graphs* (graphs without parallel edges).

Examining the two private methods, `removeFalseEdges` and `fillBackEdges`, also reveals critical information: The `Graph` class maintains only directed graphs, and simulates undirected graphs using additional edges (called both “false” and “back” edges). Algorithm writers must be made aware of this fact because the simulation of undirected graphs is not complete. For example, the result of counting the number of edges by retrieving their enumeration and querying its size is erroneous for undirected graphs.

Concept #6 connects the concepts of the `nodeList` and `edges` fields, and introduces a single method, `pack`. From its name and a glimpse of its documentation, we learn that this method is used for maintenance: It changes the indices of nodes to consecutive numbers. The fact that no other methods in the class call this method makes us conjecture that this maintenance is not mandatory.

Concept #7 connects the concepts of the `edges` and the `direct` fields, allowing us to predict (before even looking at the code) the functionality of `setDirected` and `insertEdge`. The first receives the indices of two nodes and uses the `edges` collection to add and remove the false edges used for simulating an undirected graph. The second inserts a false edge (if one is needed) in addition to inserting the genuine one.

Concept #8 defines several methods which use the three fields: `nodeList`, `edges` and `isDirected`. We can predict that the two `insertEdge` methods use the `nodeList` field to retrieve two nodes, and then call the version of `insertEdge` from concept #7.

The fact that the `setGMLvalues` method is located in this concept is important, since it suggests that the `idHash` and `lastTopId` fields are redundant: When a graph is stored in GML¹ format, the contents of these fields are not used.

¹GML (Graph Markup Language) is an XML based file format used by VGJ for storing graphs.

Concept #9 is the only concept above the bottom layer that introduces a field, `lastTopId`. As can be expected, this concept introduces the node insertion methods, which involve allocating a new identifier (using the `lastTopId` field) and then updating both data structures. The complexity in node insertion suggests again a unification of the two collections of nodes.

Concept #10 is the top-concept. In addition to the expected `copy` method, it contains six more methods which are concerned with node removal and the maintenance of groups. We learn that the removal of a node causes the removal of connected edges, and that some nodes serve as groups of other nodes.

The structured manner in which we explored the `Graph` class allowed us to detect undocumented restrictions, redundant methods, and other weak-points in the system. We also gained a better understanding of the functionality and implementation of this class.

5.2 Modified version

Although the VGJ project was discontinued, it has been adapted by many universities, and expanded for specific projects. Because it was originally designed as an applet and not as a library, many changes were made in an ad-hoc manner inside the original files.

We shall now investigate a version of the `Graph` class, which was modified and augmented by at least² one group of students at the Technion (Israel Institute of Technology). This version has as many as 9 fields and 69 methods, whereas the original version had only 5 fields and 43 methods.

By comparing the contexts for the modified class and the original class, we learn that no methods were removed and that methods which were modified access at least the same fields that they accessed before. Therefore, all the concepts of the original lattice exist in the new one and can be detected automatically. Also, note that methods of the original class which access the new fields in the modified class have relocated to the new concepts.

Fig. 7 depicts the concept lattice of the modified version of the class. In the interest of clarity, each concept in the figure contains a short description of its methods. Also, concepts which exist in the lattices of both versions are highlighted.

In comparing the lattice in Fig. 7 with the original lattice in Fig. 6, we see that the new lattice has the following nine new concepts:

- Concepts #11–14 correspond to the four new fields. Each concept contains a new field and methods for manipulating it.
- Concept #15 is an *empty concept* which does not contain any methods.³

²Over the years, a number of undergraduate programming projects at the Technion adapted VGJ for use in the research of optical networks layouts. Because no version control system was used and every group used the products of its predecessor, the revisions history of this class was lost. We obtained this version from the supervisor of these projects.

³An empty concept can be replaced with additional edges that connect every concept which dominates the empty concept directly to every concept which the empty concept dominates directly. In the lattice of Fig. 7, we can connect

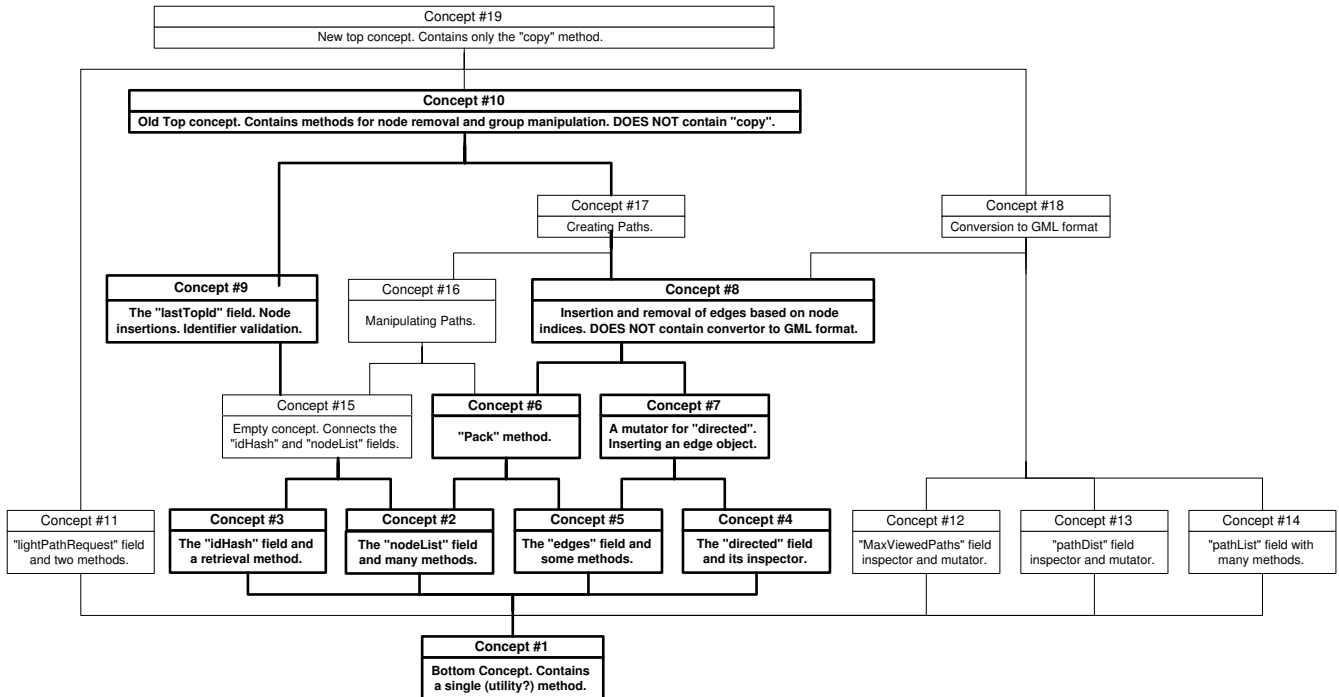


Figure 7: Simplified concept lattice of the modified `Graph` class
(Concepts which exist in the lattice of the original class are highlighted)

- Concepts #16–17 appear between the concepts of the original lattice, and contain methods for maintaining paths.
- Concept #18 connects concepts #12–14 and contains the GML conversion method which relocated from concept #8 of the original lattice.
- Concept #19 is the top concept of the new lattice and contains only the `copy` method.

Arguably, the most important difference between the two lattices is that the top concept of the original lattice is not the top concept of the new lattice. Since no method in the new version (except `copy`) uses all the fields, there is a lack of cohesion between the original fields and the new fields. This lack of cohesion suggests that `Graph` should have been subclassed, with the additions appearing only in the subclass.

Also, note that the new fields in concepts #12–14 are used in the conversion to GML whereas the new field in concept #11 is not. This difference suggests that perhaps two different groups of programmers have added these two groups of fields. Furthermore, since there is no apparent connection between the new fields in concepts #11–14 and the path manipulation methods in concepts #16–17, perhaps as many as three groups of programmers were involved.

The comparison between the two lattices allowed us to outline the significant changes and their implications without being overwhelmed by the amount of change. These results would have been difficult to achieve with a simple comparison of the source files.

both concepts #9 and #16 to concepts #2 and #3.

6 Future Research

In this work, we have demonstrated that concept lattices built upon the access relation between methods and fields can be used for understanding and inspecting a single class. Our ongoing research is currently concerned with answering the following questions:

- Can our methodology be integrated into CASE tools and code-browsers?
- Can we simplify complicated lattices further?
- What useful metrics can we calculate on these lattices?
- Can this methodology be applied to the interactive design of classes?
- What kinds of methods exist in every layer of concepts?
- Is it possible to automatically annotate methods in the lattice with the *nano-patterns* that they implement?
- Can we expand this methodology to handle multiple classes?

References

- [1] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.
- [2] David A. Watt. *Programming Language: Concepts and Paradigms*. Prentice-Hall, 1990.
- [3] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In *Ordered Sets*, pages 445–470. 1982.
- [4] B. Ganter and R. Wille. *Concept Analysis: Mathematical Foundations*. Springer Verlag, Berlin-Heidelberg, 1999.

- [5] Robert Godin and Hamed Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 394–410. ACM Press, 1993.
- [6] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. of the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.
- [7] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 99–110. ACM Press, 1998.
- [8] Gregor Snelting. Concept analysis - a new framework for program understanding. In *ACM SIGPLAN/SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–10. ACM Press, 1998.
- [9] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 300–311. ACM Press, 2001.
- [10] Alastair Dunsmore, Marc Roper, and Murray Wood. Practical code inspection for object-oriented systems. In M. Lawford and D.L. Parnas, editors, *WISE'01: Proceedings of the 1st Workshop on Inspection in Software Engineering*, pages 49–57. Software Quality Research Lab, McMaster University, Hamilton, Canada, 2001.
- [11] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice-Hall, 1992.
- [12] Tal Cohen and Joseph (Yossi) Gil. Self-calibration of metrics of Java methods. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, November 20-23 2000. TOOLS Pacific 2000, Prentice-Hall.
- [13] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
- [14] Larry Barowski and Carol McReary. Visualizing graphs with java. Reverse engineered with permission.
http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html.