Using a Molecular Metaphor to Facilitate Comprehension of 3D Object Diagrams

Brian A. Malloy
Clemson University, SC, USA
malloy@cs.clemson.edu

James F. Power
National University of Ireland, Maynooth
jpower@cs.nuim.ie

Abstract

This paper presents a strategy for the visualization of dynamic object relationships in Java programs. The metaphor of a chemical molecule is used to aid comprehension, and to help in reducing the size of the object graph. Our strategy has been implemented by dynamically instrumenting Java bytecode to collect trace data, which is then analyzed and visualized in 3D using VRML. Quantitative and graphical results are presented, based on an analysis of programs in the SPEC JVM98 and JOlden benchmark suites.

1. Introduction

The Unified Modeling Language (UML) is one of the most successful visual languages used in software development. It provides a wide range of visual representations of software, describing aspects of its design, architecture and behavior. One of the primary motivating factors behind the current design of UML 2.0 has been *model-driven engineering*, emphasizing the seamless link between visual representations and source code, and providing for the integration of UML diagrams into all stages of the software life-cycle.

The approach has already been successful with class diagrams, which describe static aspects of the software's architecture. Many design and development environments already provide for round-trip engineering, where a programmer can develop class diagrams and code in tandem. However, the integration of other types of UML diagrams, particularly those concerning the dynamic behavior of software, is not as well advanced.

The object diagram is an important UML diagram that describes dynamic aspects of the software system, and can be seen as a dynamic manifestation of the class diagram. An object diagram is basically a graph, where the nodes are objects, and the edges represent object relationships and interactions. It is usually used in design to demonstrate a typical run-time configuration, but can also be reverse-engineered from a running program as an aid to comprehension. One of the principal difficulties of visualizing and understanding object diagrams is their size. Whereas a class diagram may contain hundreds of classes, a full object diagram could easily contain millions of objects.

In this paper we seek to establish reverse-engineered object diagrams as a useful visual language for program comprehension. We focus on inter-object relationships based on field assignments, and draw an analogy with the problem of visualizing chemical structures at the molecular level. This analogy is inspired by Bergstra, who saw it as a "programming style" rather than a visualization tool [2]. The analogy is direct: classes correspond to chemical elements, objects correspond to atoms, and links between objects based on field assignments correspond to chemical bonds between atoms. We exploit this analogy by applying visualization techniques for chemical compounds to UML object diagrams.

The rest of this paper is organized as follows. Section 2 briefly reviews related work in the area of 3D software visualization. Section 3 describes the programs we analyze in this paper, and Section 4 discusses the implementation details of our approach. Section 5 demonstrates the application of our approach to programs from the SPEC JVM98 and JOlden benchmark suites, and Section 6 concludes the paper.

2. Background and related work

In this section we briefly review work relating to software visualization. Due to space constraints we do not consider 2D visualizations, and present only the most relevant work in 3D visualization. We distinguish three main types of research in this area: (1) graph-based visualizations of static software architecture, (2) non graph-based visualizations of dynamic software behavior, and (3) pedagogical visualization and animation tools.

(1) Static, graph-based approaches. There has been a considerable amount of work in the design and implementation of tools to visualize UML class diagrams. As well as

Accepted for the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)
Dallas, Texas, USA, 21-24 September 2005

straightforward representations of the relationships between packages, classes and interfaces [1], some approaches investigate the use of navigation [7], focus and context [11], or novel shapes in VRML [5]. An interesting approach uses software metrics to calculate similarity measures between classes, and uses this as a grouping criterion for positioning the classes [13]. At least two of these approaches employ the spring embedding algorithm, used in this paper, to layout their diagrams [1, 13].

It should be noted that there is a significant difference in scale between class diagrams, which rarely exceed hundreds of classes, and object diagrams, which can involve hundreds of thousands or millions of objects. One of the issues addressed in this paper is dealing with large numbers of objects by representative sampling of dynamic trace data.

(2) Dynamic, non graph-based approaches. A number of representations of dynamic aspects of programs have been proposed, and these deal specifically with the problems of representing large amounts of data. The types of visualizations include box displays [18], tree-maps [12], matrices, clusters and time charts [8] and 3D bar charts [15]. The GCspy tool, used for heap visualizations in Java, explicitly traces object behavior, producing charts showing heap activity [17].

While our work exploits a non-standard molecule metaphor to aid visualization, it is an important goal of our research to adhere as closely as possible to standard software engineering modeling structures, such as the UML. Hence, our goal has not been to create a novel form of visualization, but to explore the practical aspects of using existing standard modeling languages.

(3) Pedagogical approaches. A number of approaches and tools exist for presenting aspects of software systems for pedagogical purposes, ranging from algorithm animation [21], explaining design patterns [4], to depicting class instantiations and collaborations [20].

It should be noted that most of these approaches use custom-designed examples, and do not address the issue of scale and complexity in dealing with software systems, as opposed to pedagogical examples.

In our previous work, we have developed tools to profile running C++ applications, and to display UML collaboration and object diagrams as the program is running [14]. In this previous work, we dealt with the problem of scale by allowing user interaction, so that the information available could be filtered both at compile time and at run-time. We concluded that object diagrams were the less informative of the two types of diagram, and that further work was required to investigate effective use of object diagrams in program comprehension.

Program	Description
_201_compress	Modified Lempel-Ziv compression method
_202_jess	Expert Shell based on CLIPS
_205_raytrace	A raytracer working on a scene depicting a dinosaur
_209_db	Multiple functions on a memory resident database
_213_javac	The Java compiler from SUN's JDK 1.0.2.
_222_mpegaudio	Decompresses ISO MPEG Layer-3 audio files
_228_jack	A Java parser generator that is based on PCCTS
bh	Hierarchical force-calculation algorithm
bisort	Implementation of a bitonic sort
em3d	Propagation of waves through 3D objects
health	A simulation of the Columbian health-care system
mst	Computes the minimum spanning tree of a graph
perimeter	Computes the total perimeter of a region
power	An optimal power pricing algorithm
treeadd	Recursive depth first traversal of a binary tree
tsp	Implementation of the traveling salesman problem
voronoi	A Voronoi diagram for a random set of points

Table 1. The benchmark programs used in our study. *The seven programs in the top half of the table are from the SPEC JVM98 benchmark suite, while the ten programs in the bottom-half of the table are from the JOlden benchmark suite.*

3. The programs used in our study

The programs used in our study consist of the SPEC JVM98 [19] and JOlden [3] benchmark suites. The SPEC JVM98 suite consists of seven Java programs that are intended to represent different classes of "real world" Java applications. The JOlden benchmarks are Java versions of pointer intensive C programs, designed to exhibit a large volume of object creation. The programs in both suites are summarized in Table 1. While the SPEC JVM98 benchmark programs are more directly comparable to other studies that use Java software, we include the more synthetic JOlden programs in our study to ensure that our study scales to significant levels of object populations.

Both benchmark suites include a test harness to ensure that results from different executions are comparable. The JOlden benchmarks were run with the supplied standard parameter settings, and the SPEC benchmarks were run at size 100. Each program was run using the client virtual machine (build 1.5.0-b64) from Sun's J2SE 5.0.

A summary of some static measures of the programs is given in the first two data columns of Table 2. This data reflects the basic structure of the UML class diagram for each program, where the nodes in the diagram represent classes, and the edges represent class inheritance. As is the case for the rest of this paper, we consider only user classes (i.e. not those from the Java API), and fields that contain references to objects (i.e. not to basic types such as int etc.). As can be seen from the first column of data in Table 2, the SPEC JVM98 suite contains substantial programs, ranging from 31 to 183 classes. The level of inheritance use is also significant, peaking at 125 inheritance edges for the *_213_javac*

	Class Diagram		Objects	
Program	Classes	Inherit	Created	Alive at HWM
_201_compress	31	3	335	63
_202_jess	167	20	3883600	45270
_205_raytrace	44	10	5039697	143335
_209_db	22	1	15651	15461
_213_javac	183	125	2008795	349812
_222_mpegaudio	70	18	999	999
_228_jack	74	31	154822	15532
bh	9	2	7397764	52552
bisort	2	0	131071	130785
em3d	4	0	4009	4004
health	8	0	1025342	129225
mst	6	0	1050624	1034471
perimeter	10	3	905838	449960
power	6	0	23605	22398
treeadd	2	0	1048575	1032533
tsp	2	0	16383	16381
voronoi	6	1	2241912	567398

Table 2. Measures of the benchmark programs. *The first two data columns show the number of classes and inheritance edges in the class diagram. The last two data columns show the number of objects created during the program's run and the number of objects alive at the program's high water mark.*

program. In contrast, the JOlden suite consists of relatively small programs, with few classes and little use of inheritance.

A summary of some dynamic measures of the programs is given in the third and fourth data columns of Table 2, in order to give an overview of the complexity of the task involved in processing the object diagrams. The third data column gives the total number of objects created during the running of each program. The fourth data column gives the number of objects alive at the program's *high water mark*. The high water mark for a program is that point during its run at which the maximum number of objects are alive, and is commonly used in garbage collection studies [10]. It should be noted that the fourth column is dependent on the performance of the garbage collector, and thus will vary across implementation platforms. As can be seen from Table 2, the JOlden programs do indeed exhibit high rates of object creation and, more importantly for our study, high rates of object survival.

4. Implementation strategy

Figure 1 presents an overview of the system used to collect the data, which consists of two main components. The first component is a profiler written in Java, that analyses and instruments the class files of the input program as they are loaded by the Java Virtual Machine (JVM). The second component is a suite of programs written in C++, that builds the object diagram, selects a representative sample, and generates the VRML representation.

The core technology used to track object creation and deletion as well as field assignments, is the *java.lang.instrument* framework, incorporated into version 1.5 of the JVM. This framework allows the user to intercept classes as they are loaded by the JVM, and we use it both to generate a class diagram and to instrument the class file. The class diagrams are extracted by analyzing the class files, and are generated as a graph in Dot format. The instrumentation code, written with the aid of the Apache Byte Code Engineering Library (BCEL) [6], is inserted in constructor and finalizer methods as well as around field assignment instructions, and has the effect of generating a trace file tracking these events when the program is run.

The remaining analysis is done off-line, based on analyzing the trace file generated during the program run. First, the trace file is analyzed to select those objects and fields alive at the program's high water mark, and a summary trace is produced. This reduced trace only contains information about those objects and field assignments extant at the high water mark, but preserves the order of object creation and field assignment. The reduced trace is then used to build a graph representing the object diagram, and to compute the overall frequency of edge occurrences. From this, a representative sample is selected, the 3D co-ordinates are computed, and a VRML version of the diagram is generated. These processes are discussed in the following subsections.

4.1. Sample selection

As can be seen from Table 2, the size of the full object diagram at the high water mark varies from program to program, but is quite large in many cases. Object diagrams containing more than a few hundred nodes present problems for the layout algorithm, for the VRML browser and, most importantly, for the comprehensibility of the generated diagram. Hence, rather than trying to view the whole object diagram, we choose to select a smaller sample of the diagram. From experimentation, a figure of 250 edges was selected as yielding the largest object diagram that could be viewed with ease.

Our approach for selecting a sample is based on two observations. First, programs tend to exhibit repetitive behavior, with the relationships between objects strictly controlled by the type system. Hence, it is reasonable to assume that selecting a sample might yield representative results. Second, studies of the connectivity of heap objects and of the performance of garbage collectors indicates that object connectivity correlates strongly with object lifetimes [10]. This leads us to select a sample based on grouping events that occur together, rather than the more complex, and more computationally expensive approach of traversing the object diagram looking for a cluster of representative objects.

Having decided on a sample size of 250, each consecutive sequence of 250 field assignments is examined, and the

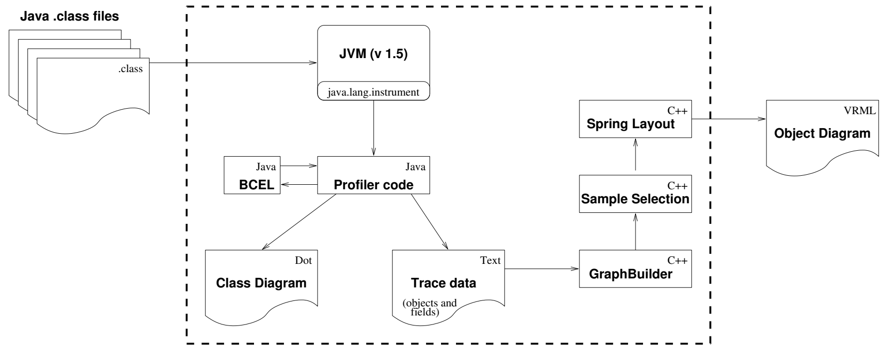


Figure 1. System overview. *The main input to the system is a set of Java .class files, which are run on an instrumented JVM. A class diagram and trace file are generated by our profiler. The trace file is then used as input to our graph layout code, which generates a VRML representation of a representative sample of the object diagram.*

frequency of occurrence of each field is calculated for that sample. The "goodness of fit" of this sample is calculated by comparing it to the frequencies for the fields across the whole program, using the following formula:

$$\chi^2 = \sum_{i \leq N} \left(\frac{(O_i - E_i)^2}{E_i} \right)$$

Here, N is the number of different fields in the program, E_i, is frequency of field i in the whole object diagram, and O_i is the frequency of field i in our chosen sample.

The χ^2 value is calculated for all possible samples of 250 sequential edge assignments, and the sample with the lowest value of χ^2 is chosen as the representative sample.

4.2. Graph layout

In order to calculate the co-ordinates of the objects in the 3D visualization, we use a simple spring embedding algorithm commonly used in chemical applications. The basic algorithm is due to Eades [9], and our implementation is adapted from one by Mutton [16].

The algorithm works by initially assigning random positions to the objects, and then iteratively assigning new positions based on calculating the attracting and repelling forces on these objects. Two nodes a distance d apart attract each other with a force of $2\log(d)$ for each edge between them, and repel each other with a force of $\frac{1}{\sqrt{d}}$.

The algorithm is iterative with the force on each object being calculated and applied on each iteration. For the diagrams in this paper, a value for M, the number of iterations, of 1000 is used.

Once the positions have been calculated, the VRML version of the object diagram is generated. Objects are represented as spheres, with the color indicating the class to which they belong. Optionally, a number representing the unique class identifier can also be displayed. Both the color, and the class identifier, are the same as those used in the class diagram. Fields between objects are represented as cones, with the apex pointing from the owning object, towards the field value. To ease comprehension, multiple edges between two objects in the same direction are not distinguished, but the presence of edges in opposite directions can be determined.

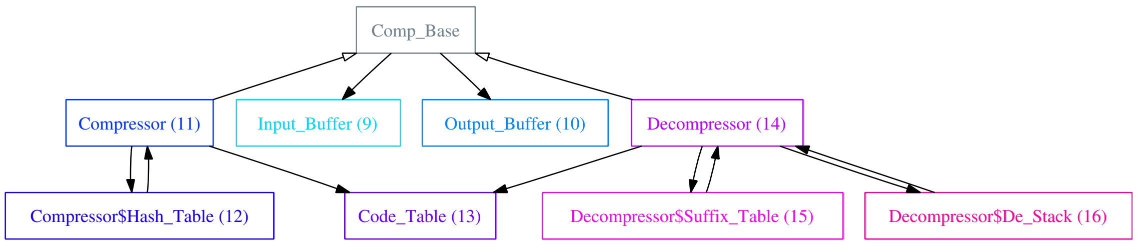
5. Results

In this section we present some of the results obtained by generating class and object diagrams for the programs in the benchmark suites. Due to space constraints we can only present a sample of the diagrams here.

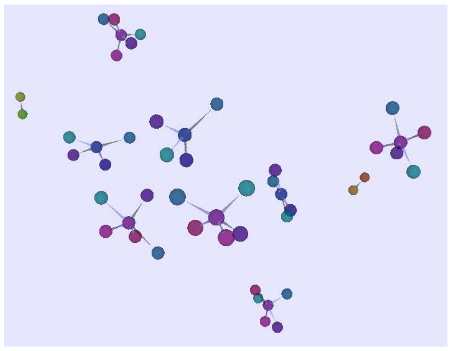
5.1. Simple "molecular" structures

As a simple example of the visualization of class diagrams and object diagrams, we present the results for the smallest program, *_201_compress*, in Figure 2. The class diagram is shown on top, in Figure 2 (a), and three views of the object diagram are shown on the bottom of the Figure. Figure 2 (b) is an overview of the object diagram, which is relatively simple, with only 10 non-trivial clusters.

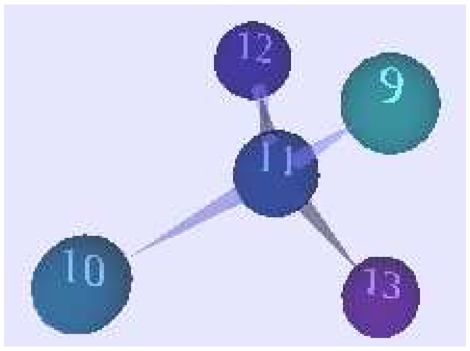
Figure 2 (c) and Figure 2 (d) show a close-up of two commonly occurring structures. Figure 2 (c) is a cluster that occurs three times, with an instance of class 11 at the center, and containing instances of classes 9, 10, 12 and 13. Figure 2 (d) is a cluster that occurs five times, with an instance of class 14 at the center, and containing instances of classes 9, 10, 13, 15 and 16. Referring to the class diagram, we can see that these two structures represent two different behaviors of the program: Figure 2 (c) represents compression, centering on class *Compressor*, and Figure 2 (d) represents decompression, centering on class *Decompressor*.



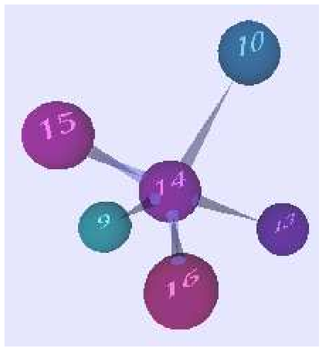
(a) Class diagram (classes representing scaffolding code are not shown)



(b) Object diagram - full view

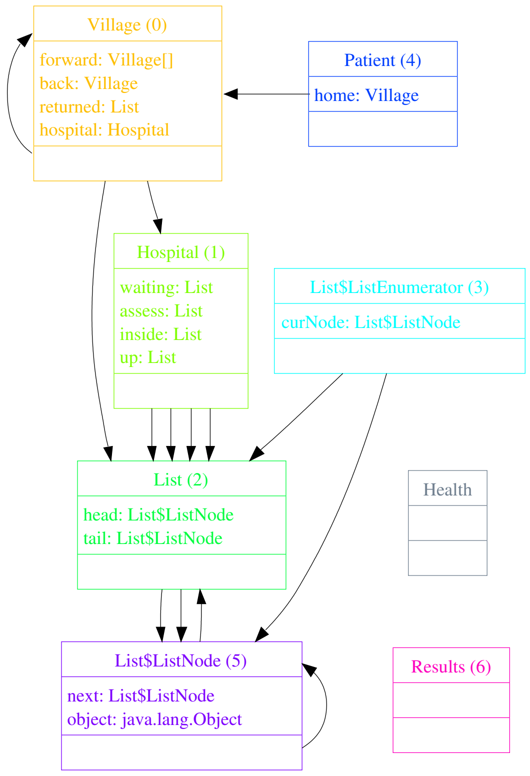


(c) Close up of compression objects

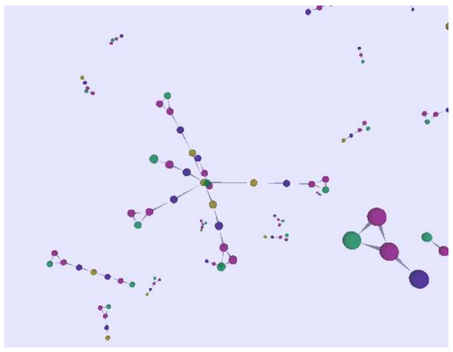


(d) Close up of decompression objects

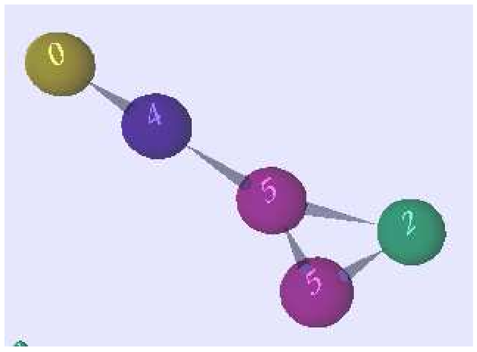
Figure 2. The class diagram and three views of the object diagram from the *_201_compress* program.



(a) Class diagram

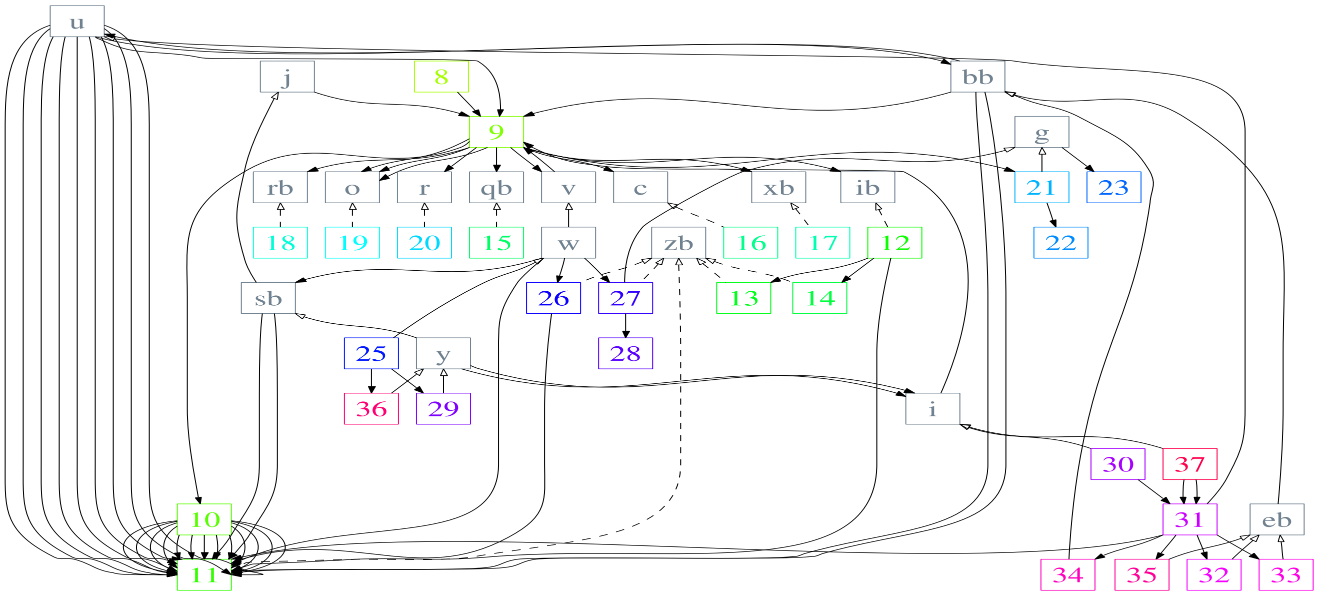


(b) Object diagram - excerpt of full view

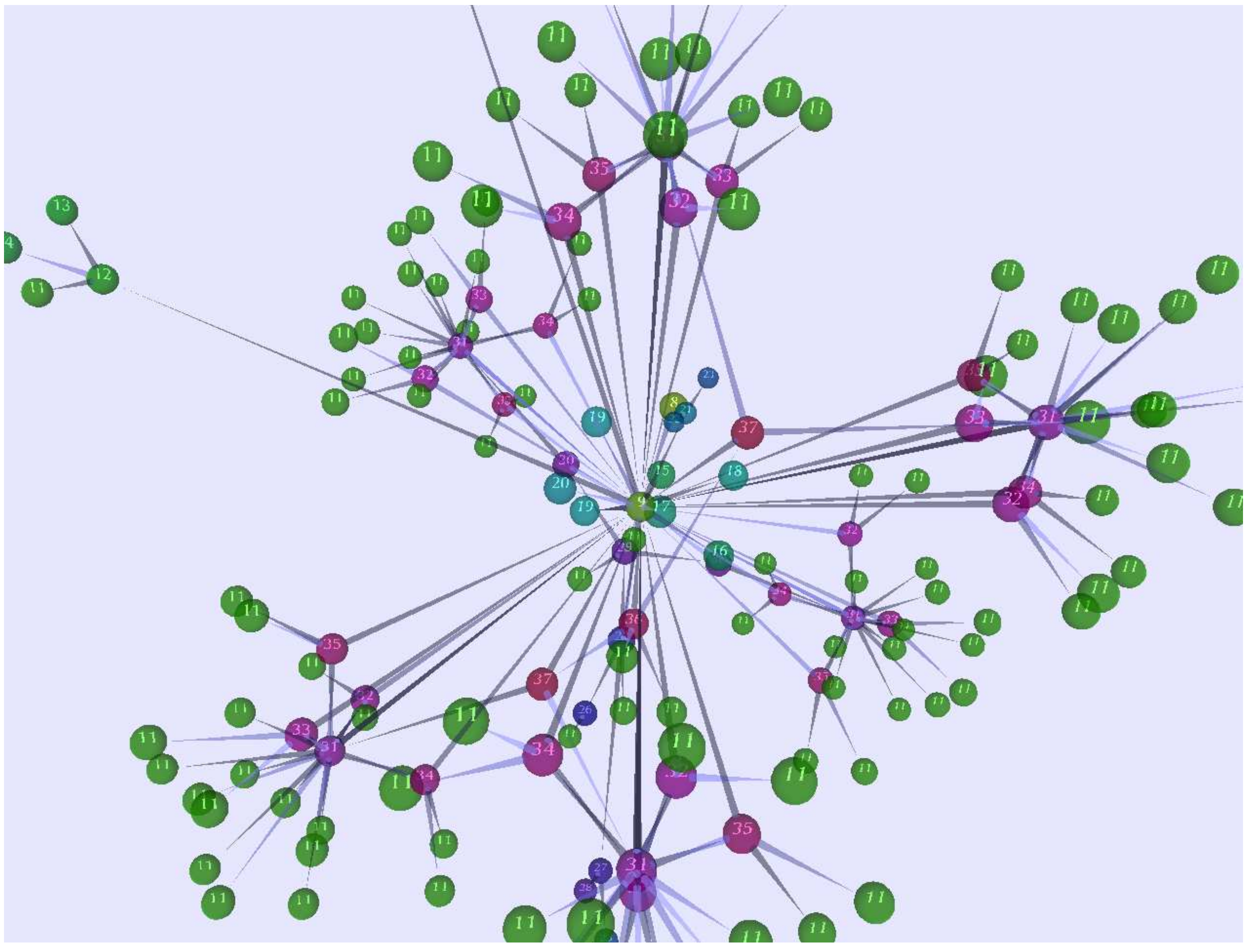


(c) Close up of a common structure

Figure 3. The class diagram and two views of the object diagram from the *health* program.



(a) Class diagram: some classes have been omitted, and numbers are used instead of class names.



(b) The main cluster from the object diagram

Figure 4. Class diagram and object diagram from the *_222_mpegaudio* **program.**

A slightly more complex example is shown in Figure 3, which shows the class diagram and views of the object diagram for the *health* program. Most of the structures in Figure 3 (b) are composed from, or are fragments of the structure shown in Figure 3 (c). From the class diagram, we can see that this represents a patient object, with an associated village object, and contained in the node of a list object.

While the object diagrams in Figures 2 and 3 must, of necessity, be consistent with their corresponding class diagrams, it is worth noting that each object diagram delivers significant extra information. In particular, the nature of the relationships between clusters of objects are easily discernible in the object diagram, but not from the class diagram.

5.2. A more complex example

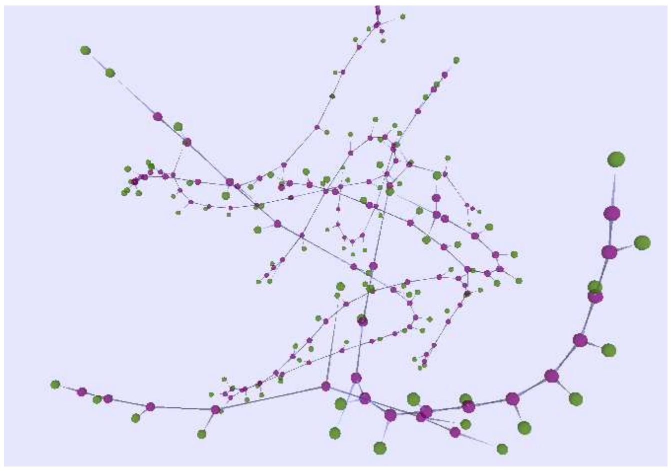
Figure 4 presents a more complex example, representing the class and object diagram of the *_222_mpegaudio* program. Apart from having a relatively complex class diagram, as shown in Figure 4 (a), the class names in the distributed version of the program have been obfuscated, which poses a further barrier to comprehension. Because of this, we represent the class names in Figure 4 using numbers, since the names themselves have little meaning.

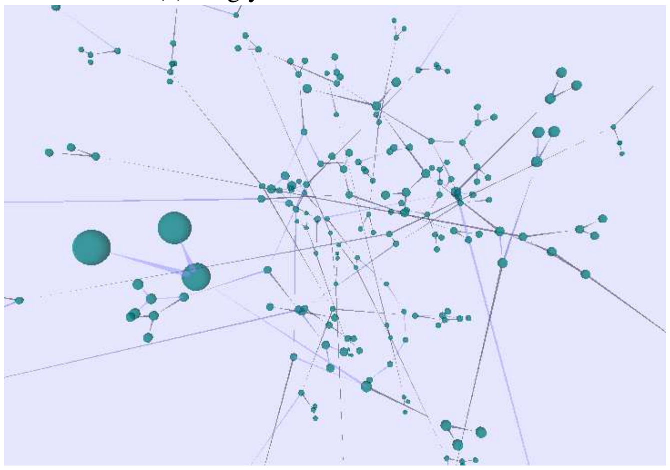
The object diagram is shown in Figure 4 (b), and consists of a single large cluster of objects, with an instance of class 9 at the center. Around this are an instance each of classes 15 through 20, colored light blue, and emanating from the center node are six sub clusters. Each of these consists of five objects, instances of classes 31 through 35, colored pink, with a total of 18 instances of class 11, colored green, emanating from these.

It is interesting to note the contrast between the class diagram in Figure 4 (a) and the object diagram in Figure 4 (b). Both of the classes 9 and 11 appear to be of central importance in the class diagram, but the object diagram makes the differing role of these classes clearer. It is also interesting to note that the similarity of the two groups of classes, 15 through 20, and 31 through 35, are mirrored in the object diagram.

5.3. Visualizing data structures

Several of the programs in the JOlden benchmark suites have relatively few classes, but implement basic data structures such as lists and trees. Figure 5 shows some of the examples of these data structures as they occur in the object diagrams of the programs. We refer to these as *user-defined* data structures to distinguish them from the containers defined in the Java API, which are not at present included in our visualization.


(a) Singly Linked List from *bh*


(b) Binary Tree from *bisort*

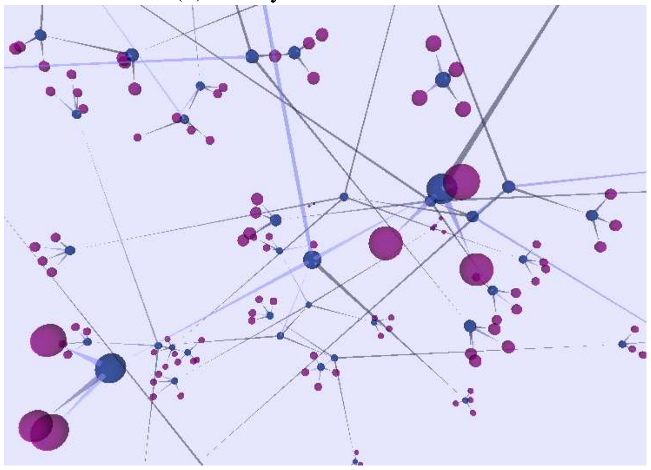

(c) Quad-tree from *perimeter*

Figure 5. User-defined data structures from three programs in the JOlden suite. *In each case, the program's object diagram clearly exhibits the influence of the underlying data structure.*

Figure 5 (a) shows the object diagram from the *bh* program, which consists of a singly linked list. The “spine” of the list is clearly visible as a chain of purple objects, with the contents field shown as the green object connected to each list node. This is a particularly simple example, since no two nodes share their contents, thus giving a linear structure.

Figure 5 (b) is taken from the object diagram for the *bisort* program, and exhibits the structure of a binary tree. The 3D representation of the tree is perhaps less intuitive than the usual 2D representation, but a clear pattern of one or two children for each node is discernible.

Figure 5 (c), taken from the object diagram for *perimeter*, shows a slightly more complex example of a tree. This diagram represents a quad-tree, where each node can have up to four children. Further, the internal nodes in the tree, colored navy, can be seen to be of different type from the leaf nodes, colored purple.

6. Conclusions

In this paper we have described a technique for extracting object diagrams from Java applications, and for visualizing these diagrams in 3D using VRML. We have applied our technique to programs from the SPEC JVM98 and JOlden benchmark suites, and have demonstrated how the derived diagrams can aid program comprehension.

We identify three main contributions of this work. First, we demonstrate the use of object diagrams in software visualization and comprehension, rather then in design, their usual area of application. Second, we use a molecular metaphor to guide our strategy for 3D visualization and layout, allowing us to harness existing techniques from the domain of chemistry. Third, we present a sampling strategy that reduces the size of the object diagram to ease the cognitive burden on the user, and demonstrate that the results add to our understanding of the software system.

Our work to date has specifically excluded Java API classes, with the result that collections of data often appear as fragmented molecules in our visualizations. An important area of future research is to investigate suitable visualization schemas for containers, such as (non user-defined) vectors and hash tables, so that these can be represented within our framework.

References

[1] K. Alfert, A. Fronk, and F. Engelen. Experiences in 3-dimensional visualization of Java class relations. *J. of Integrated Design and Process Science*, 5(3):91–106, Sept 2001.

[2] J. A. Bergstra. Molecule-oriented programming in Java. *Information & Software Technology*, 44(11):617–638, Aug 2002.

[3] B. Cahoon and K. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 280–291, Sept 2001.

[4] M. Callaghan and H. Hirschmüller. 3-D visualisation of design patterns and Java programs in computer science education. In *Conf. on Integrating Technology into Computer Science Education*, pages 37–40, Aug 1998.

[5] K. Casey and C. Exton. A Java 3D implementation of a geon based visualisation tool for UML. In *Conf. on Principles and Practice of Programming in Java*, pages 63–65, June 2003.

[6] M. Dahm. Byte code engineering library (BCEL), version 5.1, Apr 25 2004. http://jakarta.apache.org/bcel/.

[7] T. A. Davis, K. Pestka, and A. Kaplan. Kscope: A modularized tool for 3d visualization of object-oriented programs. In *Intl. Workshop on Visualizing Software for Understanding and Analysis*, pages 98–103, Sept 2003.

[8] W. DePauw, R. Helm, D. Kimelman, and J. M. Vlissides. Visualizing the behavior of object-oriented systems. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, Oct 1993.

[9] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[10] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Intl. Symposium on Memory Management*, pages 36–49, June 2002.

[11] T. Jacobs and B. Musial. Interactive visual debugging with UML. In *ACM Symposium on Software Visualization*, pages 115–122, June 2003.

[12] J. A. Jones, A. Orso, and M. J. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[13] C. Lewerentz and F. Simon. Metrics-based 3D visualization of large object-oriented programs. In *Workshop on Visualizing Software for Understanding and Analysis*, pages 70–77, June 2002.

[14] B. A. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualization*, May 2005.

[15] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *ACM Symposium on Software Visualization*, pages 27–36, June 2003.

[16] P. J. Mutton. 3-d VRML graph drawing package in Java. http://vrmlgraph.i-scream.org.uk/, 18 Dec 2000.

[17] T. Printezis and R. Jones. GCspy: an adaptable heap visualisation framework. In *Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 343–358, Nov 2002.

[18] S. P. Reiss. Visualizing Java in action. In *ACM Symposium on Software Visualization*, pages 57–65, June 2003.

[19] SPEC. SPEC releases SPEC JVM98. Press Release, Aug 19 1998. http://www.specbench.org/osg/jvm98/press.html.

[20] U. Thaden and F. Steimann. Animated UML as a 3D-illustration for teaching OOP. In *Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts*, July 2003.

[21] M. E. Tudoreanu. Designing effective program visualization tools for reducing user's cognitive effort. In *ACM Symposium on Software Visualization*, pages 105–114, June 2003.