KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs

Timothy A. Davis
Department of Computer Science
Clemson University
tadavis@cs.clemson.edu

Kenneth Pestka
Department of Computer Science
Clemson University
kapestk@clemson.edu

Alan Kaplan
Panasonic Technologies
Princeton, NJ
kaplana@research.panasonic.com

Abstract

Visualization of software systems is a widely used technique in software engineering. This paper proposes a 3D user-navigable software visualization system, termed KScope, that is comprised of a modular, component-based architecture. The flexibility of this construction allows for a variety of component configurations to validate experimental software visualization techniques. The first iteration of KScope is described and evaluated.

1. Introduction

Software visualization has become an important means by which software engineers can study and understand complex software systems at any stage during the software lifecycle – from initial development to legacy code maintenance. Currently, a popular 2-dimensional (2D) approach to software visualization is represented by the Unified Modeling Language (UML). The application of 3D visualization systems to software engineering is challenging, as in 2D representations, since the software components are abstractions that have no immediately recognizable shape or substance and choices must be made as to the physical object used to represent each software component.

While both 2D and 3D representations can take advantage of shape recognition, symbol set knowledge, and the color awareness abilities of viewers, 3D systems add depth, motion, distance, transparency, animation and spatial orientation as data transmission tools. These additional attributes allow a larger set of data values to be incorporated into a single view. Accordingly, a large amount of information can be communicated more quickly and with a higher assimilation rate [10]. The ability to navigate a 3D visualization space further allows a software engineer to transition from one view to another in a seamless manner.

As a means of testing various possible aspects of a 3D visualization system, a modularized tool, termed KScope, has been designed and a prototype implementation has been developed. The advantage of a component-based system is that it facilitates the creation of experiments in which some components are fixed, while others are altered or replaced in order to compare the efficiencies of various configurations. For instance, parsers for various languages might supply the definition of scene objects while the rest of the components are held fixed in one configuration, and thus allow a realistic evaluation of the degrees of variation in the output based on the language input.

The development of the KScope visualization system is planned as an iterative process. The first iteration defines the following five dimensions as specified in [8] :

- *task* – provide an analysis of java programs
- *audience* – researchers in the implementation of visualization systems
- *target* – Java class files as the data source
- *representation* – primitive visual objects in the analysis
- *medium* – a navigable 3D visualization with a system architecture that allows for the easy substitution of alternate components.

Subsequent iterations will expand the definition of these dimensions.

Several key issues arise when formulating a software visualization system: finding a suitable symbol set for representing abstract program concepts, placing these objects in 3D space to enhance understanding and minimize confusion, ensuring the system scales across systems of varying size, and finally, selecting some form of criteria for evaluating the effectiveness of the visualization.

2. Related Work

Research in the area of 3D visualization over the last few decades has covered many areas. Several studies [4] [10] validate that visualization of software systems increases the ability to acquire knowledge of a software sys-

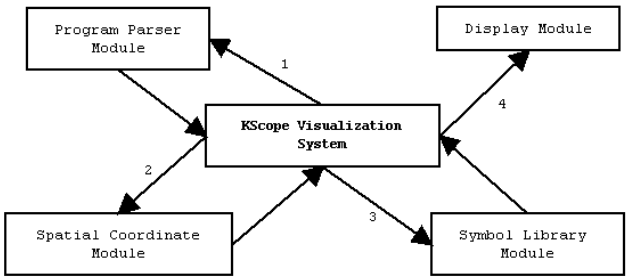

Figure 1: KScope architecture

tem, and that 3D visualizations are more effective than 2D ones in transmitting information to a software engineer. The exploration of visualizing C++ programs as a means of increasing program comprehension is illustrated by [4]. This work explores the basic visual symbol set of objects and relationships in a 2D system and justifies visualization as an important learning tool in understanding software.

A 3D representation increases the software engineer's information perception over that of a 2D representation [10]. Additionally, combining motion (e.g., rotation of the scene) and a 3D stereo view (e.g., through navigation) is a useful visualization technique to aid in understanding the structure of object-oriented code. In many instances, 3D visualization overcomes the limitations of 2D visualizations and in minimizing user confusion and increasing data comprehension [7].

The use of 3D visualization may be especially useful in understanding class structure of Java programs, as evidenced by the J3Browser tool [1]. Here, transparency, depth, ordering in space, and motion are used to convey a large amount of information in a highly effective manner. Of course, object placement and structure of the visual scene are significant in the overall success of the system.

Symbol sets and object metaphors are also significant in creating effective software visualizations. One symbol set, the "Software World" metaphor [5], uses a cityscape based on classes represented as districts within a city, with each district composed of buildings that represent class methods. This type of symbol set can be effective, but in all cases, the symbol set should reduce the complexity of the concept being visualized [6].

More recent customized symbol sets, such as the static shaded 3D symbols in [3], are also effective in user understanding as compared to the standard UML class diagram symbols. The symbols in this set are constructed using a basic symbol alphabet known as *geons* [2] and outperform the UML set in all the illustrated experiments.

In evaluating the effectiveness of 3D visualization systems, two distinct elements should be considered: representation of objects and the mode of visualization [11]. The first element involves the representation of abstract concepts as physical entities, and covers the symbol set used in the visualization. The second set of criteria deal with the visualization itself. These criteria will be used in assessing the effectiveness of KScope.

3. Implementation

3.1 Architecture

The implementation of the KScope tool is based on the following of components (see Figure 1):

- KScope Visualization System – acts as the main driver and calls the other modules in the order shown along the arrow connectors
- Program Parser Module – extracts meaningful information about the structure of the program under inspection
- Spatial Coordinate Module – determines the location in 3D space of each of the scene objects
- Display Module – maintains the interactive 3D environment on a chosen visual system

Java, a reflective language, was chosen as the language to be analyzed by this first iteration of the KScope visualization tool. The extensive Java Byte Code Engineering Library (BCEL) [??] is freely available and allows for easy implementation of the parsing stage**.** The initial symbol set is made up of simple primitive objects including cubes, pyramids, and lines that can be rendered quickly in the display component. KScope uses the Java3D graphics library because of its cross-platform capabilities and the ease of configuring across various display devices.

Our sample test case is based on classes and interfaces coded in such a way as to illustrate five UML relationship types represented within the main class, and within the classes and interfaces used by the main class. Classes are named to show their relationships with the class under analysis. For example, the parent class of Child_Main_1 class is named "Parent." The types of relationships illustrated include: inheritance, association, dependency, composition, and implementation. *Inheritance* in Java is based on extending the functionality of a parent class, while *implementation* is based on an implementing class defining the methods of the interfaces. There is an *association* relation between classes when a reference to a class object existing outside the class under analysis is passed as an argument to a class constructor. A *dependency* exists where a class object is an argument to a method of the class under analysis. A *composition* relation exists when a class object is created within the class under analysis and has a lifetime less than or equal to the lifetime of the class (e.g., a class object as an attribute of a class).

Figure 2 shows our test case as analyzed by the 2D visualization system, Together Version 6.0 [9]. The entire static class representation uses standard UML notation to represent the various relations.

Figure 3 shows the same example test case as analyzed by KScope. In KScope, coloring is used as a significant element in defining symbols. A multicolored cube represents the main class (i.e., the class containing the main method) under analysis. The cube shape is used to indicate a class, while a pyramid indicates an interface. The dark blue shaded cubes represent what are called terminator classes, which are those classes defined outside the directory of the class under analysis, including the standard Java library classes and other predefined libraries. Terminators are primarily used to limit the extent of the analysis. The light purple pyramids represent interfaces, while the light green pyramid represents a terminator interface. The color of the connecting line indicates the relationship of the connected objects: association is red, dependency is blue, composition is magenta, implementation is black, class inheritance is green, and interface inheritance is yellow.

Two forms of text-based information are available to the user: class and interface names, and additional information displayed by selection with the left mouse button. For classes and interfaces in the current directory, a BCEL-derived analysis appears; in the case of terminators, the appropriate JavDoc appears.

3.2 Object Placement

Within KScope spatial orientation is used to indicate the direction of relationships. Class inheritance proceeds upwards, as indicated by the green vertical lines connecting classes. An interface is placed on the horizontal plane of the class that implements it, while interface inheritance, like the class method, is indicated vertically. The composition, dependency and association relations are placed below the main class with appropriately colored connecting lines. Terminators are place above the main class.

The placement of each type of object is determined by separate placement algorithms. The main class is always placed at the origin of the display universe (0, 0, 0). Classes that are part of an inheritance hierarchy are placed above their respective child class (i.e., parent classes are always higher than their children). Related classes are set in place based on conical calculation; that is, a count of the related classes is used to divide a circle into equal arcs with classes placed at the end point of each arc. The entire circle is displaced on the negative Y axis based on the generation of the parent class. The radius of a placement circle is based on the number of objects in the set and the number of generations below the

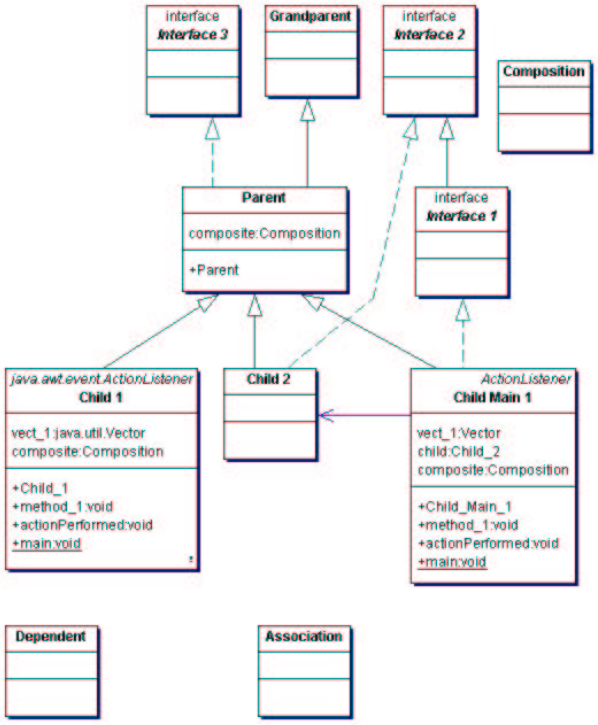

Figure 2: Together 6.0 class analysis

main class. The terminator classes are placed in a similar manner with displacement on the Y-axis in a positive direction. Interfaces are offset in the positive X direction with inheritance in a vertical displacement.

3.3 User Navigation

User navigation is performed via the keyboard (see Table 1).

Table 1 Keyboard navigation

Key	Action
←	move viewpoint left
→	move viewpoint right
↑	move viewer toward objects
↓	move viewer away from objects
page up	move visual objects up (viewpoint down)
page down	move visual objects down (viewpoint up)
=	return to start view

4. Results

Figure 3 shows the initial view of the test case under analysis. The user can select (through a drop-down menu) three other views: related classes, interfaces and their related classes, and terminators, as shown in Figures 4, 5 and 6, respectively. In all of the views, the entire software representation initially rotates at a constant rate.

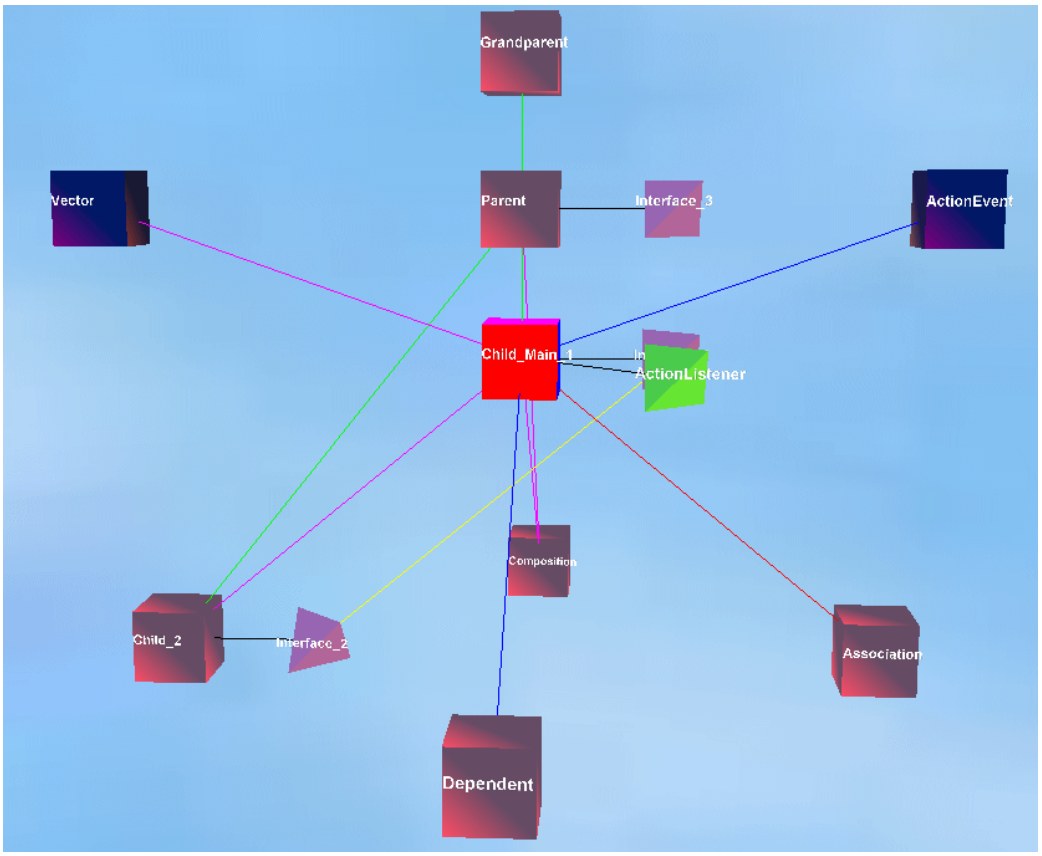

Figure 3: KScope: all views

The user has the option to start or stop the rotation as desired. Note that Figures 2 and 3 analyze the same system.

In each view an object can be inspected by left clicking on it. This brings up a pop-up window (see Figure 7) with a textual description of the class or interface, as explained in Section 3.1. Various details of the class are listed within this text area. Additional items can also be included; the items shown were selected for brevity.

A complex system can create a complex visualization, as shown in Figure 8 (KScope's self analysis); therefore, the system includes a user choice of level of detail. The user can right click on a class to change the view to one that contains the class and terminators related to that class (see Figure 9).

4.1 Evaluation

The evaluation of KScope is based on the criteria discussed in Section 2 [11]. First, the abstract representation contained in the system is considered:

- individuality – color is used to express the individuality of the types of objects
- distinctive appearance – classes are distinguished from interfaces by 3D shape
- high information content – colors, shape and text are used to present data to the user
- low visual complexity – primitive shapes, such as the cube and pyramid, keep visual complexity low
- scalability of visual complexity and information content - KScope performs well for small to medium sized programs, but the scalability to large programs has yet to be demonstrated
- flexibility for integration into visualizations - the flexibility of the overall system is reduced when color is applied as a distinguishing characteristic to the object symbols (i.e., color is no longer available as a characteristic of some other aspect of the analysis); however, such tradeoffs are often necessary
- suitability for automation - selection of simple primitives require low processor time to display

When considering the visualization criteria as pertaining to KScope the following is noted:

- simple navigation with minimum disorientation – the small number of keyboard and mouse commands makes KScope's user interface extremely simple
- high information content – large amounts of information is presented in each view
- well-structured with low visual complexity – ar-

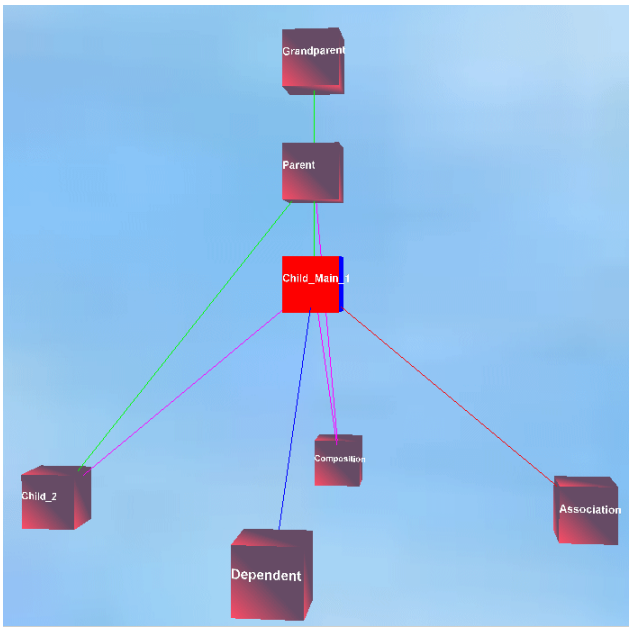

Figure 4: KScope: "Relationships" view

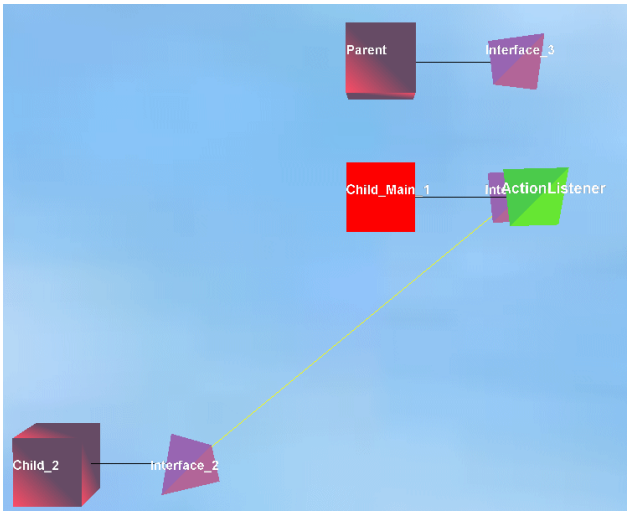

Figure 5: KScope: "Interfaces" view

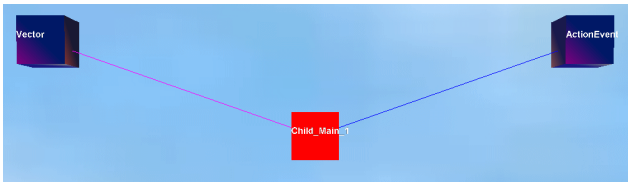

Figure 6: KScope: "Terminators" view

rangement of the objects in hierarchical cone-like formations represents a structuring of the visualization; however, the arrangement of objects with a multitude of connector lines does lead to a complex visualization; rotation of the scene, as well as selectable views for reduced object sets and relationships, gives the viewer an enhanced understanding of the virtual space and increases the ability to compre-

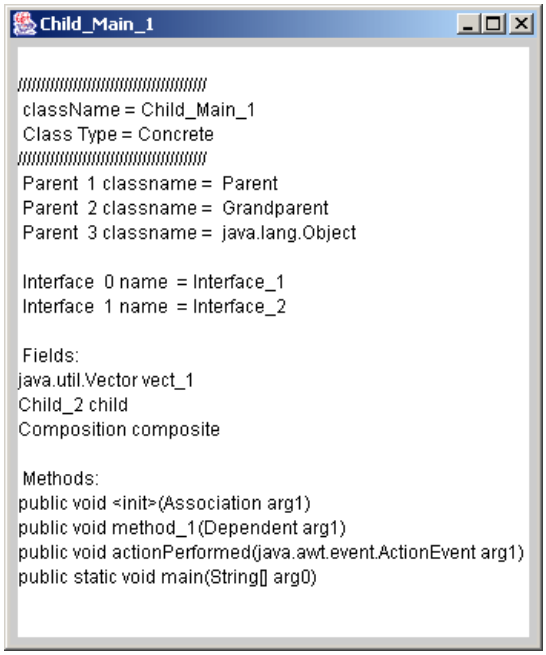

Figure 7: Class pop-up

hend the relationships of the objects

- varying levels of detail - the user's ability to select a class and view the related terminals is a direct application of this criterion
- resilience to change – as the views are changed, relationships of objects is maintained in either a subtractive or additive selection of views
- good use of visual metaphors – no attempt was made to find visual metaphors
- approachable user interface – the limited controls and simple selection process of the user interface satisfy this criterion
- integration with other information sources – the text-based representation of the object integrates the 3D view with another source of information
- good use of interaction – user interaction with the objects is shown through the ability to navigate the visual space and to select alternate sources of information
- suitable for automation – the system is fully automated in the generation of the visualization.

5. Conclusion and Future Work

Future work will focus on additional spatial coordinate selection algorithms, symbol sets, and language parsers. Further additions to the symbol library could be achieved by extending the "geon" symbol alphabet [3] into a virtual environment and thus, greatly expand the number of geon-defined objects and the variety and number of relationships expressed in a single view.

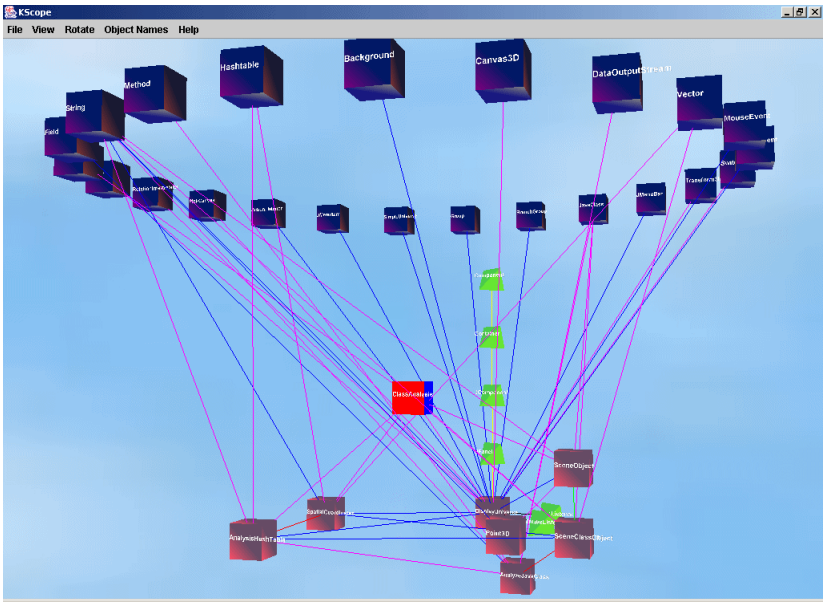

Figure 8: KScope analyzes KScope

The modular structure of KScope lends itself to various extensions. Improvements to the system would involve additions to the current number of alternative modules. By increasing the variety of components, such as additional selections of the type of spatial coordinate placement algorithms, and additional symbol sets for the symbol library module, the tool will be made the basis for a series of experiments as to the quality of information presented to the user and the user's ability to comprehend the information.

Overall KScope presents the visualization of software in an interesting and visually stimulating manner. The colors, spatial placement, alternate views, multiple methods of presenting data, and viewer interaction create an effective educational environment for the user. Increasing the functionality of the system and experimenting with different configurations will build on the strengths illustrated by KScope.

6. References

[1] K. Alfert and F. Engelen, "Experiences in 3-Dimensional Visualization of Java Class Relations," *Transactions of the SDPS*, September 2001, Vol.5, No. 3, pp 91-106.

[2] I. Biederman, "Recognition-by-Components: A Theory of Human Image Understanding," *Psychological Review*, Vol. 94, No. 2, 1987, pp 115-147.

[3] P.Irani, C.Ware and M.Tingley, "Using Perceptual Syntax to Enhance Semantic Contents in Diagrams," *IEEE Computer Graphics and Applications*, (in Press)

[4] D. F. Jerding and J. T. Stasko,"Using Visualization to Foster Object-Oriented Program Understanding," *Technical Report GIT-GVU-94-33*, July 1994.

[5] C. Knight and M. Munroe, "Virtual but Visible Software," *Visualization Research Group*, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, UK. 1998.

[6] C. Knight and M. Munroe, "Comprehension with[in] Virtual Environment Visualization," *Visualization Research Group*, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, UK. 1999.

[7] H. Koike, "The Role of Another Spatial Dimension in Software Visualization," *ACM Transactions of Information Systems*, Vol. II, No. 3, July 1993 pp 266-286.

[8] J. I. Malectic, A. Marcus and M. Collare, "A Task Oriented View of Software Visualization," *VISSOFT 2002*, June 26, 2002.

[9] http://www.borland.com/together/index.html, 2003.

[10] C. Ware, D. Hui and G. Franck, "Visualizing Object Oriented Software in Three Dimensions," *CASCON 93 Conference Proceedings*, Toronto, Ontario, Canada, October, 1993, pp 612—620.

[11] P. Young and M. Munroe, "Visualizing Software in Virtual Reality," *6th International Workshop on Program Comprehension: IWPC'98.*

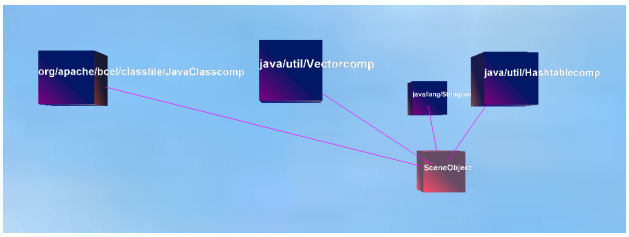

Figure 9: Level of detail