To appear at the SDPS Journal of Design & Process Science, Vol 5, 2001.

Experiences in 3-Dimensional Visualization of Java Class Relations*

Klaus Alfert
Alexander Fronk
Chair for Software-Technology
Dortmund University
D-44221 Dortmund
email: {alfert,fronk}@ls10.cs.uni-dortmund.de

Frank Engelen
Java Competence Center
SerCon Düsseldorf
40474 Düsseldorf
email: Frank.Engelen@sercon.de

October 24, 2001

Abstract Java software provides a vast amount of information about class and interface relations. Inheritance- or *uses*-relations of large software systems lay great demands on being able to overview the scene. Class browsers may help to master the information, although visualization is usually limited to two dimensions.

We analyze the benefits of 3D presentation and discuss experiences with our visualization tool `J3Browser`. The tool realizes these benefits and some selected visualization techniques within the Java context.

This paper leads a step towards a CAD-like design of Java software in 3D space.

*Remark: Should be viewed on color printouts or on color screens.

1 Introduction and Related Work

Visualization and graphic-oriented methods are common in software engineering.

In most cases these methods use 2D graphics. The use of 2D drawings is inherited from traditions in engineering, where they are working well. The great advantage of 2D drawings is that they only require paper and pencil. This fact is so important, that more complex notations like the Booch Method (Booch, 1993) have been abandoned now, and its successor, UML (Scott and Fowler, 1997), uses a much simpler notation instead.

Today, engineering sciences often use 3D CAD systems, but three dimensions are rarely used in the visualization of large computer software systems. In contrast to the products constructed in classical engineering, the software artefacts are abstract as they do not have a substantial physical corpus. No natural 3D appearance of software artefacts exists, thus making it difficult to find suitable visual concepts. In addition, the visualization of large systems raises many problems due to the limitations of the display in space and resolution.

3D is sometimes used in visual languages and in program visualization. In the tradition of the Balsa System (Brown, 1988) for algorithm visualizations, some approaches exist. For example SAM (Geiger et al., 1998) supports visual programming in 3D, Seity (Chang et al., 1995) employs a 3D direct manipulation interface of SELF objects. They primarily focus on programming-in-the-small and the visualization of dynamic sequences of events in the running program.

An additional aspect is shown in VRCS (Koike and Chu, 1998). This system presents the history of a software system in terms of versions, variants and configurations in 3D. But neither static structures nor dynamic properties, except for the system's decomposition into files, are considered.

In ArchView (Feijs and Jong, 1998), the software architecture, i.e. the structure between modules, is presented in 3D. Each module is realized as a cube or cylinder. Typical relations like *uses* or *calls* are pipes with arrows depicting the direction of the relation. The reduction to relations between modules allows ArchView to present large software systems written in a procedural language. Just a few relations suffice to build the module structure of such a software system, but expressing the structure of an object-oriented software system requires many more relations.

This fact is put to use in NestedVision3D (Ware et al., 1993). Object-oriented software is three-dimensionally displayed as a graph of related cubes the nodes of which may be refined by further graphs and are thus nested within the cubes. Edges relate arbitrary software constructs and can be faded out. This approach features the idea of hypergraphs, a well-known construction from graph theory. In contrast to our approach, NestedVision3D uses nested cubes in form of directed graphs only. Integration of different visualization techniques to enhance structural comprehension of information does not take place.

One formal model regarding the structure of a system is the syntax graph. Graph-based languages like UML consider some parts of this syntax graph and visualize them in two dimensions. In our approach we combine this kind of visualizing syntax graphs and the benefits of 3D visualization for large software systems' structures. As a starting point, we consider the prevailing static class relationships. We chose Java (Gosling and Arnold, 1996) as an example language because its object-

oriented properties are well-designed and the language is well known. As we are interested in a software engineering point of view, we do not consider algorithmic details and aspects of programming-in-the-small. Thus, we propose a more abstract look at software systems than is usual for software visualizations, yet our approach still retains comparability to other design notations.

We aim at making 3D capability more convenient by supporting viewer comprehension and overview of Java software. We do not merely convert a syntactical UML-like notation into 3D icons. As the main emphasis, we analyze some properties of 3D-modeling and show, how the third dimension can be employed efficiently to solve our problem. Koike (1993) used this idea to build a C++ class browser, but he only arranges 2D planar graphs in 3D. Unfortunately, this does not exploit the full range of the 3D possibilities.

For the sake of this paper, we select a few properties of 3D spaces. They cover the motion of objects and their transparency covering or revealing other objects behind them. Positioning objects and the effect of depth are on focus as well. In section 2.1 we discuss these aspects of three-dimensional visualization. Section 2.2 looks at several recent techniques, which translate the aspects considered into action.

Java offers different class relations like inheritance, association, local classes or packages, all worth to be visualized. Unfortunately, these relations require different properties and techniques in three-dimensional space to be visualized appropriately. Section 3.1 and 3.2 discuss some Java concepts and show, how the above-mentioned techniques can be combined soundly and properly integrated into the Java context. Section 3.3 discusses our tool for visualization of Java software, followed by a discussion of the used spring model in sec. 3.4.

We finish our paper with a conclusion and a short look on future work.

2 Analyzing 3D Space

We want to sketch how we combine requirements of a proper visualization of Java software with three dimensional properties focusing on efforts to find possible combinations of these two worlds.

2.1 Properties of 3D Space

Discussion of 3D space leads to mathematical and geometrical topics; physical assertions can also be stated. For us, none of these issues were sufficiently convincing to make 3D really work, failing to capitalize on real benefits of 3D space. We were too beguiled into attempting an imitation of “real” physical space. All experiments ended in the assertion that 2D would have done equally well. The only exception was a spring model of how to fix and move objects in space, which turned out to produce renderings with less intersections of elements in 3D. So, we wanted to find out how one can make specific use of 3D in a way which is not, or at least much less, available in 2D. Therefore, we abandon paper and pencil freeing ourselves from the restriction of producing flattened 3D, a somewhat dead still life.

We decided to concentrate on the differences between 2D and 3D, homing in on specific 3D aspects of displays or (non)-euclidian projection spaces, generating a *virtual landscape*. To make things clear, we

pick out some aspects and show their effect in 3D, which is much weaker in 2D.

Those aspects were found in

- *transparency of objects*, revealing the notion of "inside" contrasting strongly to uncolored things in 2D (see figures 1 and 2).

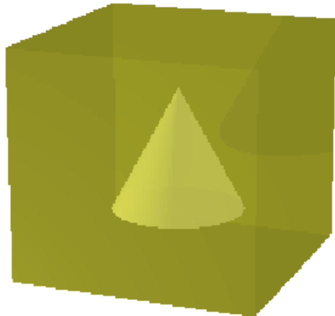

Figure 1: Transparency in 3D space

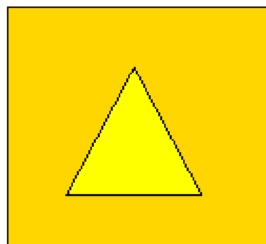

Figure 2: Transparency in 2D space

- the *effect of depth*, which we found more suitable to use in a 3D space (see figures 3 and 4).
- *ordering in space*, a specific meaning of which can be expressed and comprehended much better than in 2D (see figures 5 and 6).

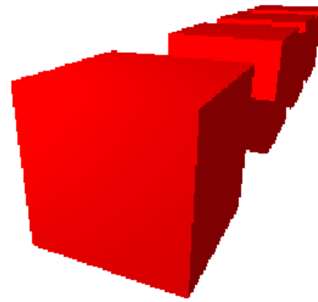

Figure 3: Effect of depth in 3D space

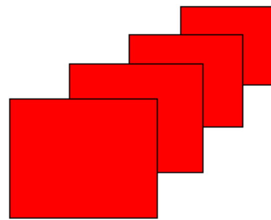

Figure 4: Effect of depth in 2D space

- *motion*, which we attach great importance to. Examples are rotation of cone trees (see sec. 2.2), or changing the observer's viewpoint continuously.

In contrast to motion, *animation* in the sense of program visualization by Brown (1988) renders dynamical changes of program states within an execution, but we do not focus on this topic. Animation as a special effect highlighting some information is nice to have but can be done in 2D as well. Therefore, we do not regard animation in any sense as an essential 3D property.

Next, we look at possibilities to structurally order objects in 3D space.

tain information without losing its context.

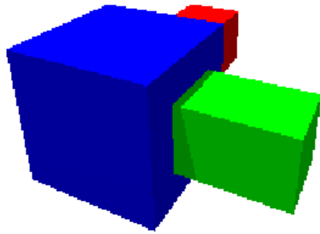

Figure 5: Ordering in 3D space

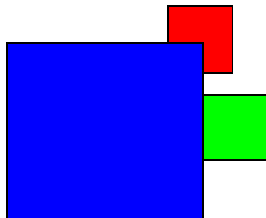

Figure 6: Ordering in 2D space

2.2 Geometric Arrangement in Three Dimensions

When we consider different objects in a space, we can think about their arrangement, i.e. their respective positions, and their relations, i.e. their connections to each other. One can find several approaches in the literature:

Cone Trees (Robertson et al., 1993) relate objects in a tree-like manner (see figure 7). The root of each subtree forms the top of a semi-transparent cone. The root's direct children are arranged on a circle line building the cone's base. This structure is applied recursively. Individual rotation of each cone allows focusing a certain information without losing its context.

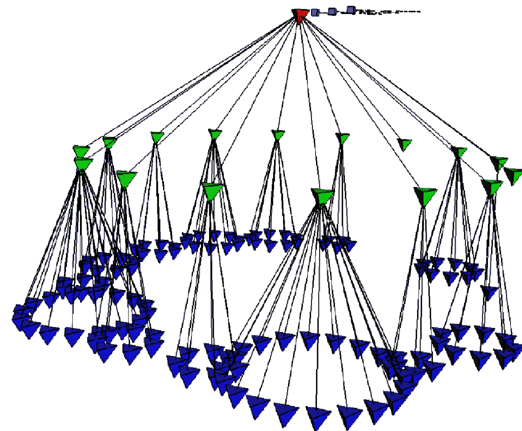

Figure 7: A cone tree

Information Cubes (Rekimoto and Green, 1993) glue together related information (see figure 8). Walls as used in Koike and Chu (1998) give chance to shape a room. Even criteria concerning the content of data can be chosen as a projection base. Such cubes may contain arbitrary visualizations, in particular sub-cubes easily accessible through transparent walls.

Information Landscapes use the metaphor of a landscape. As an example, in figure 9 the file system navigator (SGI Corp., 1992) shows file systems as landscapes of pedestals connected by wires. Such landscapes are characterized by planarity, but perspective distortions allow a broad view even without fully using 3D space.

Figure 8: A famous information cube

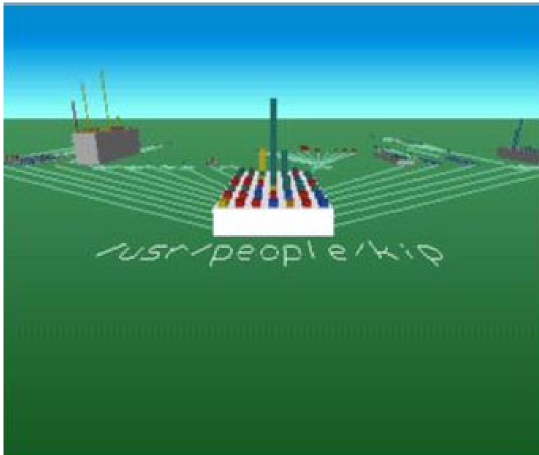

Figure 9: An information landscape

These are only a few techniques to get order into a space. Of course, they can be combined with each other, endeavored to use what we selected as aspects before. Several applications of these techniques can be found in the literature, Young (1996) gives a good overview.

3 Java and 3D Space

We will now apply the discussed properties and visualization techniques to Java software after a short presentation of its static structure. We close with a discussion of our visualization tool `J3Browser`.

3.1 The Static Structure of Java Software

Java software is statically structured by packages, interfaces and classes (Gosling and Arnold, 1996). A type in Java is understood either as an interface, as a class or as a simple type. The latter is not of interest in this paper.

Packages combine sets of classes, interfaces and packages into a strict hierarchy. Packages introduce namespaces for their entities.

Interfaces define types and consist only of signatures of methods and constant attributes. As they do not have any implementations, multiple subtyping between interfaces is allowed.

Classes implement types defined by interfaces or other classes. As usual, classes can be abstract, i.e. implementation can be deferred to subclasses. Only simple inheritance is allowed between classes, but a class can implement several interfaces at once.

Inside a class, local classes can be defined. The scope of a local class is restricted

to the embedding class. The use of a local class outside of the scope of the defining class is possible only if the local class extends or implements classes or interfaces accessible outside. Additionally, it is possible to create anonymous classes without a name.

Another static relation between classes is association realized by attributes of a related type. Besides associations, the *uses* relation may be established implicitly by parameters, return values or local variables of methods as usual. More explicitly (but only as syntactic sugar), there is the need to import classes and interfaces from other packages.

In Java, we have four different kinds of accessibility modes for methods, attributes and local classes, and two different kinds for non-local classes and interfaces. These modes restrict the possibility to connect classes and interfaces with each other similar to scoping rules.

In summary, we have two strict hierarchies (packages, classes), an acyclic directed graph (interfaces), two directed graphs (association and *uses* relations), and the restriction by accessibility modes. These different relations are heavily intertwined in real world software systems.

3.2 Experiences using 3D

Our experiences are based on experimental visualizations of source code of some non-trivial Java software systems. We analyzed the Java API of JDK 1.1.8, the `J3Browser` itself, and the `AD1300` system, a project at our chair (Alfert et al., 1999). These systems are large enough to serve as a good analysis basis for evaluating the applied visualization techniques (compare table 1).

The geometric arrangements presented in sec. 2 have in common that every arrangement is only suited to show one kind of relationship at a time. Some of these arrangements can only present tree-like relationships. In Java software different kinds of relationships coexist and are not generally tree-like. This shows that it is impossible to select one arrangement technique to depict simultaneously all relationships in a given Java software. But simultaneousness is desirable, because separated views for different kinds of relationships would burden the viewer with mentally integrating them to an overall picture. Thus, our approach for visualizing Java software in 3D space strongly focuses on concurrent display of different kinds of relationship by using different arrangement techniques in one diagram. Even if the following discussion is structured by different kinds of relationship, simultaneousness stays a primary concern.

Figure 10 shows an example of our approach. The packages are arranged as an information landscape. Following the style of ArchView (Feijs and Jong, 1998), we use a graph-like approach with nodes, represented by boxes (for classes) and spheres (for interfaces), and vertices, represented by pipes for relations. The package membership of these classes is presented by an information cube. In the foreground, in package `facade` a class hierarchy is shown as a cone tree. In the background, additional packages are shown to illustrate relationships between these packages.

In the following we describe our experience in using different arrangement techniques for different aspects of Java software. We focus on techniques with a clear relationship to 3D space. Engelen (2000) discusses further experience with these

	JDK 1.1.8	`J3Browser`	`AD1300`
Source Files	679	190	82
Lines of Code	150,289	39,797	14,712
Elements (total)	767	260	98
Classes	609	232	80
Interfaces	134	17	9
Packages	24	11	9
Relations (total)	5,287	1,476	422
uses	2,243	611	132
inheritance	479	116	29
implements	178	40	10
other relations	2,387	709	251

Table 1: Properties of the used software systems

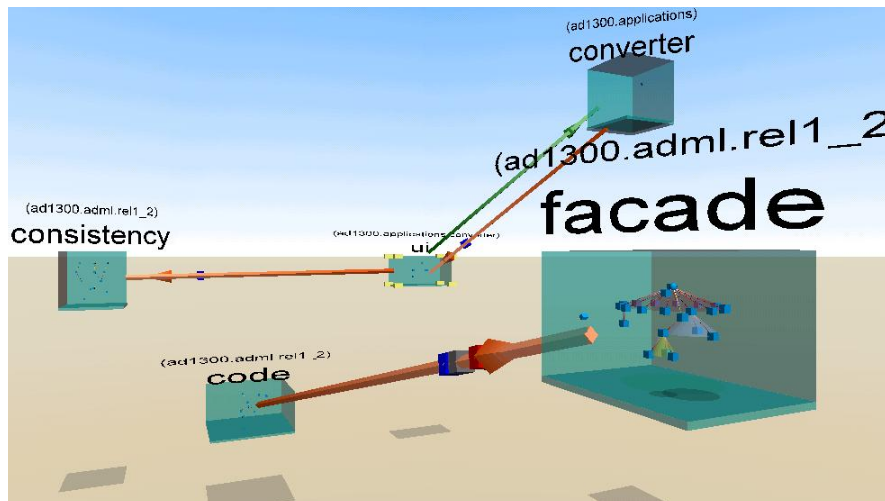

Figure 10: Exemplifying a software structure

and additional techniques not exclusively related to 3D space.

3.2.1 Relations between Classes

In Java, inheritance is a strict hierarchy suggesting the use of cone trees for its rendering (see figure 11). Cone trees allow a more compact presentation than traditional 2D trees. As an experiment, we built an expressive visualization of a class hierarchy containing all classes of the two JDK packages `java.lang` and `java.awt`. With 137 classes, it is cumbersome to display this hierarchy with traditional 2D trees. But even representing 2D trees in 3D is of benefit: the perspective distortion can be used to enlarge a few classes in focus, whereas the remaining classes of the hierarchy although smaller maintain overall context (see figure 12).

Showing additional relationships while depicting a class hierarchy – e.g. the *uses* relation between different classes –, the cone tree approach may lead to problems. Cone trees utilize space in such a way that supplementary relations may obfuscate visualization and thus destroy its structure. To avoid this obfuscation, we found it appropriate to layout the classes of the entire hierarchy on exactly one cone surface (see figure 13). The inside of this cone can then for example be used for additional classes and their relationships to classes displayed on the cone surface.

This cone layout, however, does not utilize space very well. Relaxing its layout constraints leads to leveled or top-down arrangements. With such arrangements it is easier to display additional kinds of relationships, but the class hierarchy is less emphatic. An alternative for depicting class hierarchies is the information cubes technique. We found that it gives no benefits for simultaneousness and is much less intuitive.

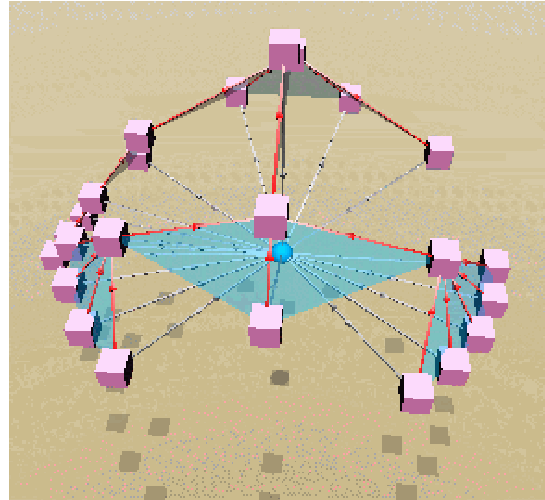

Figure 13: Cone surface ordering with an interior element

3.2.2 Relations between Interfaces

Subtyping of interfaces is not tree-like but forms an acyclic directed graph. Here, arranging interface relation top-down or leveled is a good choice.

Using 3D space for software visualization offers the advantage of depicting multiple hierarchies one behind the other. In figure 14, this arrangement is used to display implement relations between classes (shown as cubes) and interfaces (shown as spheres).

3D space helps to explicitely emphasize orthogonality of class and interface hierarchies as shown in figure 15.

3.2.3 Packages

The membership of types in packages is a good starting point to segment the visualization into multiple regions. Those can be

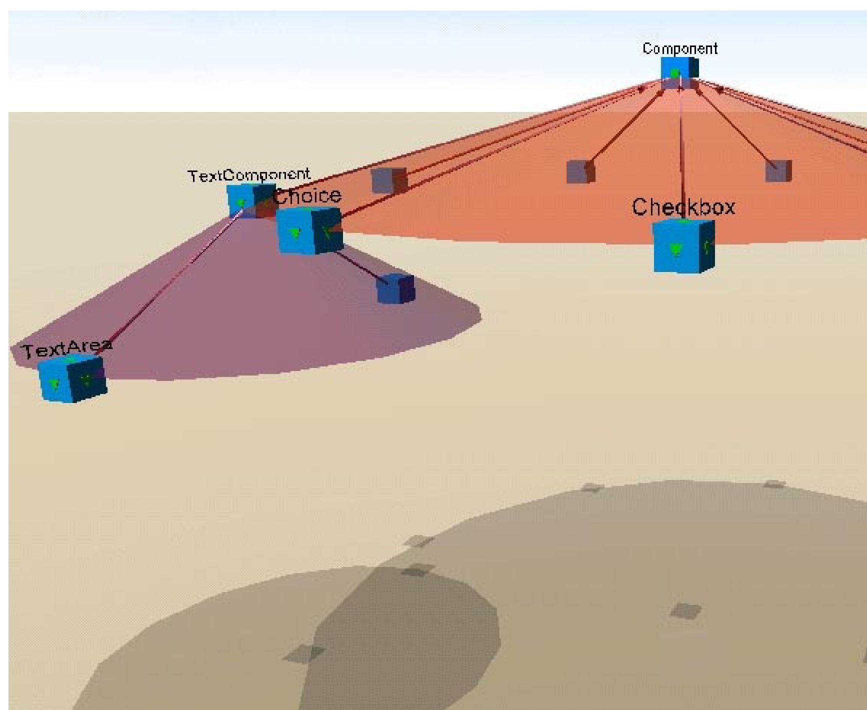

Figure 11: A class hierarchy as a cone tree

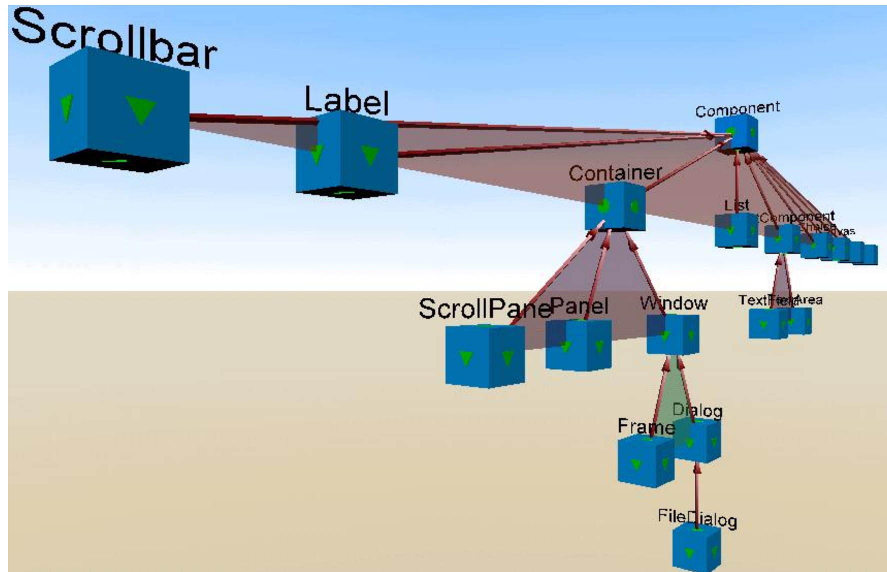

Figure 12: A class hierarchy as a wall

visited and comprehended separately. It is quite obvious to use information cubes in this context (see figure 16).

In contrast to 2D nesting, 3d has additional advantages. The observers can change their viewpoint and move into the package of interest. The transition from an overall view to a view focusing on one package is natural and seamless. In 2D systems, only zooming is offered as such a transition, thereby losing context information and orientation due to the resulting abrupt change of view. Figure 16 shows a perspective distortion to keep context information viable. To support clarity of visualization we give packages semi-transparent walls. Thus, the observer can realize the context even if their viewpoint is inside a package. Again, we use information landscapes.

Java packages can be ordered into hierarchies. We used information cubes and realized, however, that it gives no clear impression of the overall structure. If a package is located deeply within a hierarchy, the observer has to dive into the nesting. This causes packages on top of the hierarchy to get out of sight. First user tests with our prototype tool indicate that cone trees or top-down arrangements are much more expressive in such a case.

3.2.4 Associated documents

It is hard to comprehend software and its structure without additional documentation. Examples are design documents and in particular for Java HTML pages of source code generated by `javadoc`. A variety of tools exists to view such documentation. But with traditional window-oriented user interfaces, it is necessary to switch between at least two windows: one showing the visualization and the other displaying the documentation. A switch of context may demand reorientation. A closer integration of visualization and doc-

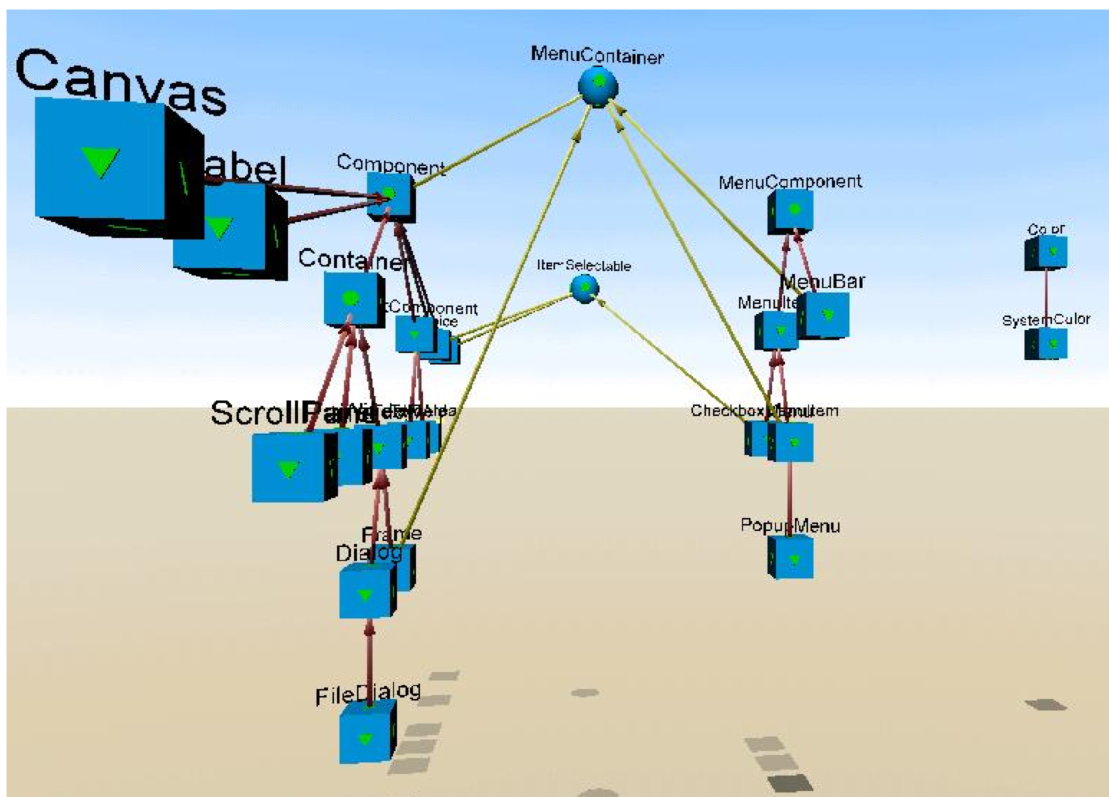

Figure 14: Hierarchies connected by interfaces

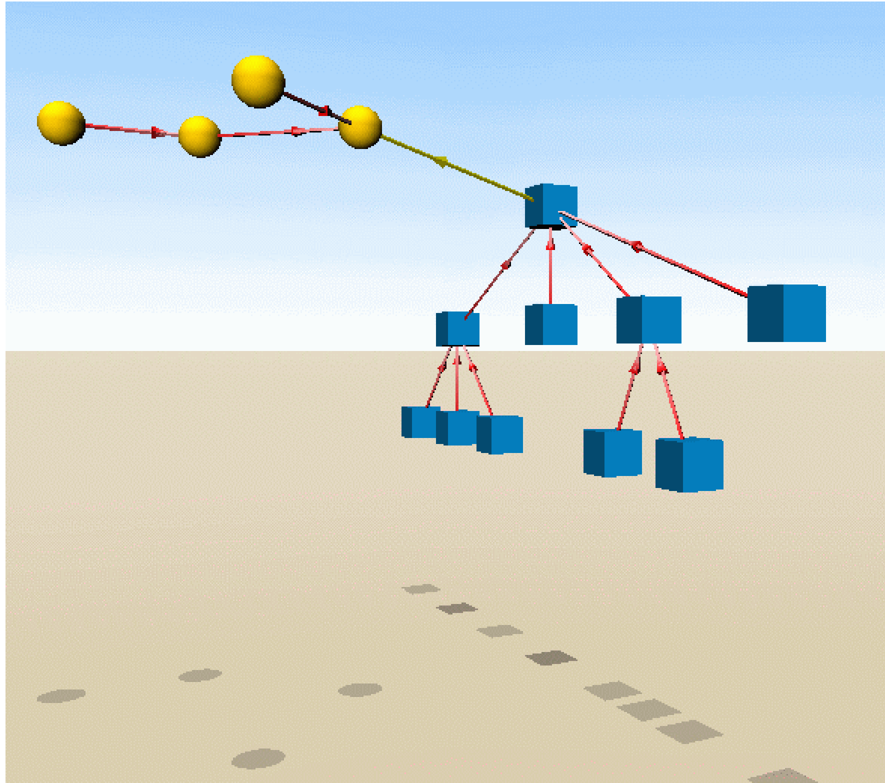

Figure 15: Orthogonal view on class and interface hierarchies

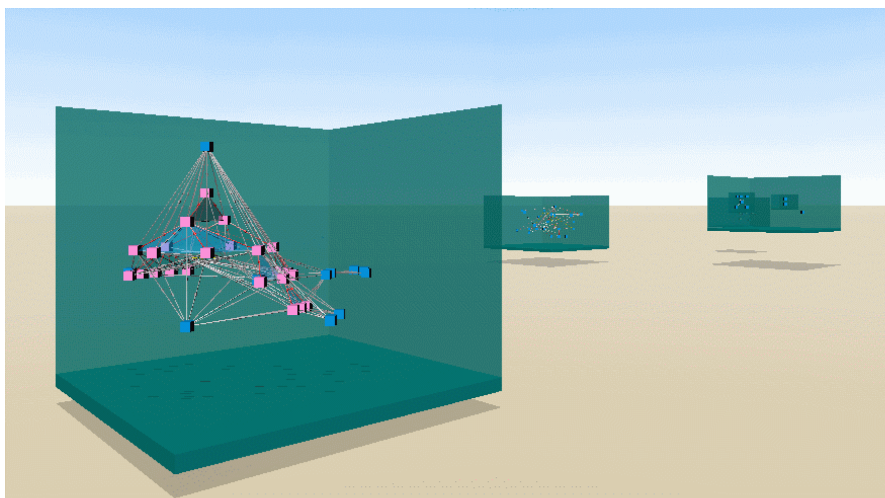

Figure 16: Nested structures

umentation is thus desirable.

We experimented with documents displayed directly onto the walls of graphical elements (see figure 17). This allows a natural integration with the overall navigation through 3D space: to achieve specific information of a class the viewer simply moves towards it and reads the documentation. This approach, however, has a great disadvantage: the viewer has to align their position to the text quite exactly. Even little distortion makes the text hard to read. It is more suitable to display the documentation in front of the visualization. It may simply be projected onto a semi-transparent screen (see figure 18), thereby the visualization remains visible. Reorientation is not required.

3.3 Tool Support

As stated above, visualization in three dimensions demands tool support, manual drawings are no longer feasible. We developed a visualization tool called `J3Browser` described in detail by Engelen (2000). The tool consists of two parts: the first part analyzes Java source code and generates a structure model. The second part is a Java applet, which interactively allows applying visualization techniques on the structure model and thereby creates a 3D model represented as a VRML scene. The scene is displayed by the Cosmoplayer plugin (Computer Associates International, 2001) within the Netscape browser. Furthermore, the applet provides higher level control mechanisms, such as setting specific parameters of visualization techniques or hiding certain types of relationships, to manipulate the scene. The entire tool is implemented in Java.

We found 3D visualization particularly useful for exploring unknown software. On the one hand, it can be very motivating to roam through information landscapes and act like an explorer. On the other hand, it is very frustrating if much effort is needed to bring up a visualization from the scratch. Thus, our tool provides an initial top-down visualization, which is extracted from a Java source code and rendered with a spring model based on the works of Ryall et al. (1997). Based on our experience, the spring constants are determined by weighted relations between elements. The spring model is discussed in more detail in section 3.4. In this way, the tool provides an immediate and intuitive first impression of the information structure.

The discussion in section 3.2 makes clear, however, that individual interests on displayed information demand an individual selection of visualization techniques. Choosing the right technique for the desired accentuation demands a lot of creativity and thus cannot be fully formalized. Hence, our tool supports a variety of techniques an observer can simply choose from and experiment with without the need to calculate important parameters by hand. More over, the tool offers useful and aesthetic renderings within the selected techniques.

We visualized different Java software systems including the Java JDK with more than 5,000 relations (see table 1). But exploring such large systems has the drawback to get lost in the visualization easily, well known as *lost-in-hyperspace* especially in the WWW. Effective usage of large visualizations demands both navigational support and the possibility to reduce the amount of information currently shown. Beneath the usual operations to navigate through a virtual landscape, such as mov-

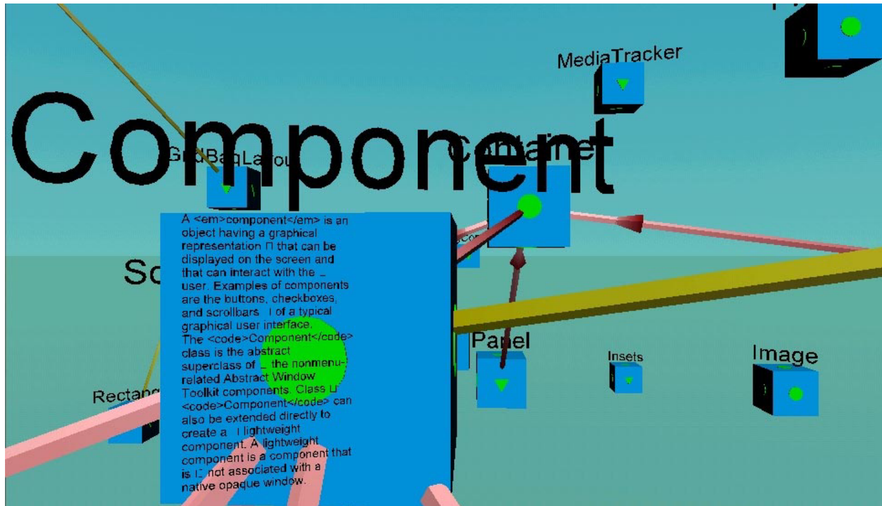

Figure 17: Wall projected documentation

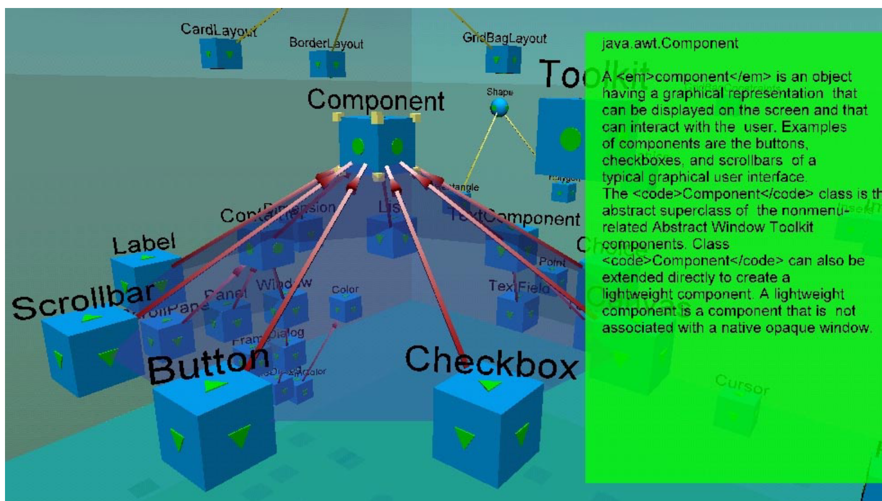

Figure 18: Screen projected documentation

ing the virtual camera, its viewing angle or zooming, our tool features different kinds of manipulations capable to reduce the information overkill. We allow selecting objects, fading them in and out, moving and transforming them in size, shape, color.

Furthermore, we use transparency to express a degree-of-interest. The scene may be as follows: the user selects an arbitrary type T. Each relation connecting T transitively to any other type is rendered with a transparency correlating to its distance to the selected type T. Thereby, types may even be rendered as isolated due to totally transparent relations if desired by the user.

Another important feature can be explained within the following scenario: due to relations within the entire system, a certain relation between types may be obscured. In figure 10, for example, the *inheritance*-relation, r, between types in different packages is not shown. To set the focus on types connected by r, the user selects any type T. Next, the spring model is exploited interactively to attract exactly the types connected with T by r and thereby concealing the desired relation temporarily: as visual effect, the related types are extracted from their packages and are arranged in near distance to T.

These features go far beyond the possibilities of traditional 3D browsers such as VRML browsers.

3.4 The Spring Model

As mentioned in the previous section, an initial visualization is generated using a spring model. The generation can be customized in different ways. To group related elements the tool allows specifying desired distances for related and unrelated elements. To achieve leveled visualizations, the level distance can be defined. Furthermore, one can provide a weighting between different kinds of relationships to change accentuation of the drawing.

In our approach, these parameters are set to values based on our experience and mapped to two parameters generally controlling the behavior of a spring model:

- the *rest length* of each spring to determine desired distances and
- the *spring constants* to determine the weighting.

Elements are the main suspension points of our spring model. Thus, we distinguish between three different types of springs(see figure 19):

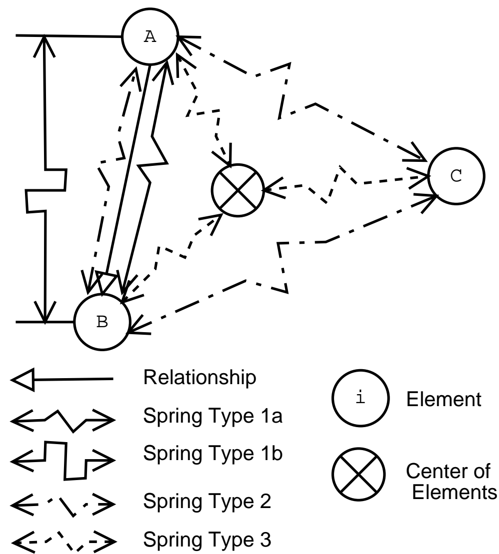

Figure 19: Structure of the spring model

1. Each relation between two elements implies two springs of different type: the first (type 1a) enforces the desired distance between the two elements and has a constant according to the weighting of the relationship. The second (type 1b) enforces top-down or leveled arrangement and is therefor only active in a vertical direction: A low constant ensures top-down visualization, whereas a high constant ensures leveled arrangements.

2. A spring (type 2) between each pair of elements ensures divergence of elements.

3. We use the initially calculated center of the elements as another suspension point to tighten a spring (type 3) to the element. Modifying constants and rest length of these springs allows building compact visualizations.

The model generated in this way is simulated by an algorithm adopted from Ryall et al. (1997). The algorithm receives as input a set of nodes, N, and a set of springs, S, and works in an iterative way: all spring forces affecting a node n are calculated and summed up for each node $n \in N$. Then, the resulting impulse on n is calculated considering damping to prevent the model from infinite oscillation. The impulses on each $n \in N$ are finally used to calculate the time-depending position $n(t)$ for each node.

In our implementation, the visualization is updated periodically, so the user can stop the simulation if he is already satisfied with the result or if he wants to refine parameters.

We also implemented a 2D variant of the described spring model in order to compare the results in 2D and 3D: the 3D spring model produced results which were far more aesthetic and clearly arranged. Disturbing edge-crossings were almost totally avoided in 3D. Trees, cone trees and other techniques can further enhance the quality of visualizations. But even with top-down and leveled arrangements only, the spring model is undoubtedly well suited for an initial overview and helps to minimize the effort for building expressive visualizations.

4 Conclusion and Further Work

In summary, we gave an impression of how we combine and integrate three-dimensional space and Java software.

Based on analyzing important properties of 3D space, we developed the 3D visualization tool `J3Browser` for static relationships in Java software systems.

The initial visualization provided by the tool is well suited for a first overview of the scene. Flexible modifications of visualization properties including our implementation of a spring model allow the observer to detect more concealed information. Our experiences show that our tool is also powerful enough to explore and comprehend even large unknown software systems quickly and easily.

Currently, we only consider sheer visualizations. Our future work aims at extending our tool towards an environment for constructive CAD-like design of software systems in 3D space. Therefore, visualizations must be made mutable. This immediately demands a suitable semantics for operations on visualizations. Our

results on abstraction levels proposed in (Alfert and Fronk, 2000) of manipulations seem to be a solid foundation for future research on the following direction: it is interesting to reveal collaboration issues between classes which require not only static but additionally dynamic relationships. Design Patterns, as introduced by Gamma et al. (1994), are a common language to express a wide range of such collaborations. Some reverse engineering tools such as the FUJABA system (Niere et al., 2001) are able to recover the use of design patterns in Java software. But, such systems use UML to visualize software structures and do not consider 3D models. Our current hypothesis is to give the arrangement of classes in space a semantics denoting roles in design patterns. For example, in the abstract factory pattern abstract factories and products are arranged on a level positioned above concrete ones. Thereby, we aim at a 3D notation for designs of software systems based on design patterns.

Acknowledgments

We like to thank E.-E. Doberkat gratefully for his helpful comments and suggestions on this paper.

References

Klaus Alfert, Ernst-Erich Doberkat, and Corina Kopka. Towards constructing a flexible multimedia environment for teaching the history of art. SWT-Memo 101, Chair for Software-Technology, University of Dortmund, September 1999. Submitted for publication. Available online `http://ls10-www.cs.uni-dortmund.de/`.

Klaus Alfert and Alexander Fronk. 3-dimensional visualization of java class relations. In *IDPT 2000 Conference - The Fifth World Conference on Integrated Design & Process Technology*, June 2000.

Grady Booch. *Object-oriented Analysis and Design. With Applications.* Benjamin Cummings, 2nd edition, 1993.

Marc H. Brown. *Algorithm Animation*. ACM Distinguished Dissertations. MIT Press, 1988.

Bay-Wei Chang, David Ungar, and Randall B. Smith. Getting close to objects. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming. Concepts and Environments*, chapter 9, pages 185–198. Manning Publications, 1995.

Inc. Computer Associates International. URL: `http://www.cai.com/cosmo/`, 2001.

Frank Engelen. Conception and implementation of a 3D class browser for Java. Master's thesis, Chair for Software Technology, University of Dortmund, 2000. In German.

Loe Feijs and Roel De Jong. 3D visualization of software architectures. *Communications of the ACM*, 41(12):73–78, December 1998.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

C. Geiger, W. Mueller, and W. Rosenbach. SAM - an animated 3D programming language. In *IEEE Symposium on Visual*

Languages, Halifax, Canada, September 1998.

James Gosling and Ken Arnold. *Java: The Language*. Addison-Wesley, 1996.

Hideki Koike. The role of another spatial dimension in software visualization. *ACM Trans. on Information Systems*, 11(3): 266–286, 1993.

Hideki Koike and Hui-Chu Chu. How does 3-D visualization work in software-engineering? Empiral study of a 3-D version/module visualization system. In *Proceedings of 20th Internation Conference on Software Engineering*, pages 516–519, Kyoto, Japan, April 1998.

Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Revocering uml diagrams from java code using patterns. In Jens H. Jahnke and Conor Ryan, editors, *Proceeding of the 2nd Internation Workshop on Soft Computing Applied to Software Engineering (SCASE)*, Enschede, The Netherlands, February 2001.

Jun Rekimoto and Mark Green. The information cube: Using transparency in 3D information visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems*, pages 125 – 132, 1993.

G. G. Robertson, S. K. Card, and J. D. Mackinlay. Information visualizing using 3D interactive animations. *Communications of the ACM*, 36(4), 1993.

K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST '97*, Banff, Alberta, Canada, 1997.

Kendall Scott and Martin Fowler. *UML distilled. Applying the Standard Object Modeling Language*. Addison Wesley, 1997.

SGI Corp. 3D File System Navigator for IRIX 4.0.1+, 1992. available online at `http://www.sgi.com/fun/freeware/3d_navigator.html`.

C. Ware, D. Hui, and G. Franck. Visualizing object oriented software in three dimensions. In *CASCON '93 IBM Centre for Advanced Studies Conference Proceedings*, pages 612 – 620, 1993.

Peter Young. Three dimensional information visualisation. Technical Report 12/96, Department for Computer Science, University of Durham, UK, November 1996. Available online at `http://www.dur.ac.uk/~dcs3py/pages/work/Documents`.