# Evaluating 3D-Visualisation of Code Structures in the Context of Reverse Engineering

Alexander Fronk *
Software Technology, University of Dortmund, Germany
Dietmar Gude, Gerhard Rinkenauer
Leibniz Research Centre for Working Environment and Human Factors, University of Dortmund, Germany

## Abstract

In reverse engineering it is a common approach to generate UML diagrams from code, capturing technical details as well as structural static relations between, e.g., packages and classes. It can be observed, however, that it is hard to depict large object-oriented systems in such a way that particulary unknown legacy code can be visually explored to comprehend it for effective maintenance. Our contribution to this task is to use relation-specific geometric arrangements in three-dimensional space. 3D space allows to display a large amount of data with optimal visual ingress, and using specific layout algorithms optimises space utilisation with respect to both exploration and interpretation of the code structures displayed. So far, we have developed a reverse engineering tool, VisMOOS, supporting these ideas by means of *3D relation diagrams*, and in the next step, both the concepts and the tool require for empirical studies to evaluate their effectiveness for understanding unknown code structures. In this project there are three different partners involved: the Chair for Software Technology of the University of Dortmund, the Leibniz Research Centre for Working Environment and Human Factors, and an industrial software developing company. Although we have different interests in the project, we follow the same goal: How can we empirically prove the effectiveness of our 3D relation diagram? In this article, we report on both 3D relation diagrams and how we are designing and conducting experiments to evaluate them.

## 1 Underlying Theory and Concepts

In reverse engineering, code structures are usually visualised using UML diagrams within a tool, for example, like OMONDO. One of the drawbacks of such diagrams heavily under discussion is scalability: large software systems are hard to depict as one single UML package and class diagram. More seriously, these diagrams need to be split up into many separate ones or into arbitrary hierarchies such that important information might be lost or simply overseen.

Reverse engineering is not a stand-alone task. It aims at visualising code such that the diagrams generated can be used for, e.g., maintaining a software system (cf. [Lano and Haughton 1993; Bax ]). Clearly, maintenance tasks require to understand the system in detail which is a non-trivial matter particularly for large and unknown legacy code (cf. [Mayrhauser and Vans 1993]). The diagrams generated by a reverse engineering tool must thus allow the observer to gather the information needed and to interpret his/her findings correctly such that maintenance tasks can be carried out both efficiently and effectively.

*e-mail: fronk@LS10.de

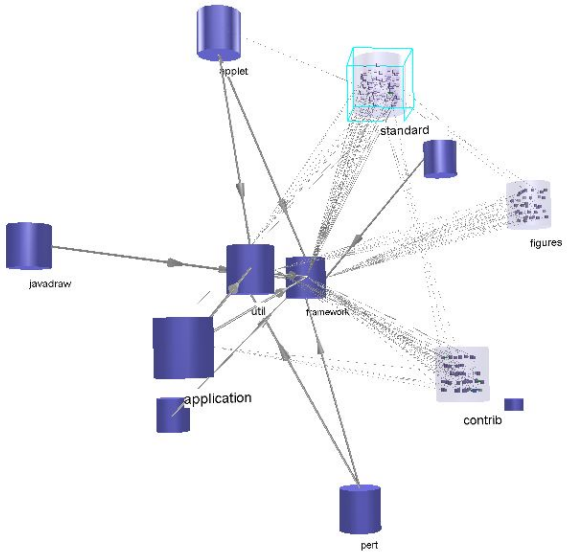


Figure 1: Package Affiliation

UML class diagram capture technical details such as attribute types and method signatures as well as structural static relations such as association or inheritance. These relations, however, are shown simultaneously with similar icons and without layout respecting, e.g., containment and tree- or star-like structures. This is what we think aggravates the comprehension of structural interrelations. Moreover, integrating class diagrams into package diagrams delivers unusable large 'paintings'. In our project, however, we consider Java software systems and stipulate that it is as necessary, however, to understand structural relations as to investigate technical details. Hence, we follow the ideas that

1. considering different geometric arrangements for different structural relations between source code entities enhances code comprehension and offers valuable insight into structures which, for example, need refactoring,

2. integrating different views, e.g., enriching package diagrams with class relations, offers important context information,

3. exploiting three-dimensional space allows for displaying more data on less space than in 2D and offers a much better visual ingress to the code structures under consideration.

We have the strong feeling that these aspects help to comprehend large and unknown software structures more quickly

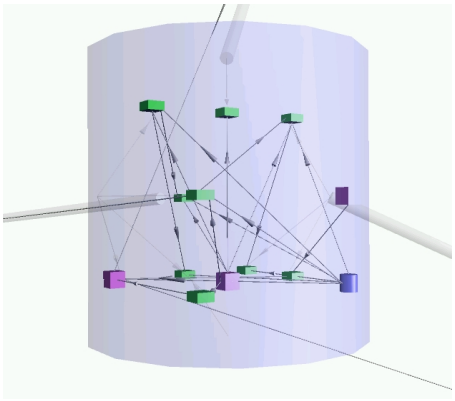

Figure 2: Package/Class Integration

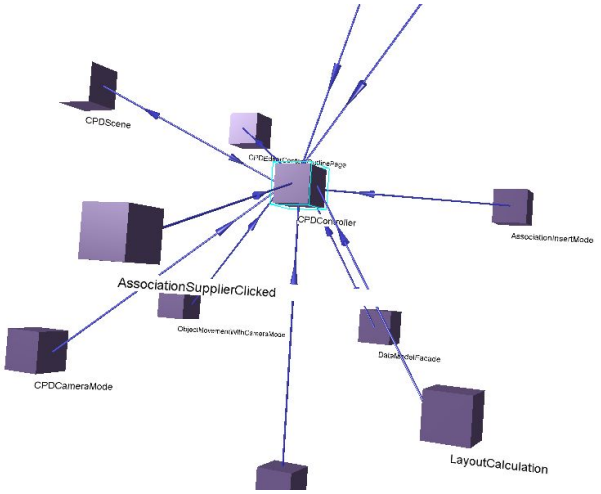


Figure 3: Class Association

and accurately than with UML diagrams alone. In detail, we have elaborated the following *3D relation diagrams* (see Fig. 1 and 2 for *package relation diagrams*, and 3 and 4 for *class relation diagrams*) to highlight different relations between Java entities by means of different geometric arrangements (cf. [Alfert et al. 2001; Rohr 2004]). That is, we have assigned well-known three-dimensional visual concepts like tree-like structures or semi-transparent information cubes [Rekimoto and Green 1993] to Java entities such that we can highlight source code relations with respect to their structural properties:

- *Package affiliation diagrams* show packages as cylinders connected by pipes. A pipe from package `A` to package `B` is drawn if in `A` there is a class or sub-package referring to a class in `B`. Simultaneously, the user may access the interior of a package revealing classes, interfaces, and sub-packages and their associations which are thus displayed within the context of their package affiliations.

- *Class association diagrams* refer to interfaces and classes alone leaving their package affiliations away. Using force-directed layout, entities with many associations in between are placed closer together than those with less or none associations.

- *Class hierarchy diagrams* represent inheritance information in a tree-like manner. With a grid-layout, the subclasses of a class are arranged on a grid underneath their common superclass. Simultaneously, interfaces and their inheritance hierarchies drawn under forced-directed layout enrich the hierarchies such that the observer may instantly capture, firstly, which classes implement which interfaces and, secondly, which interface hierarchies are in use within which hierarchies.

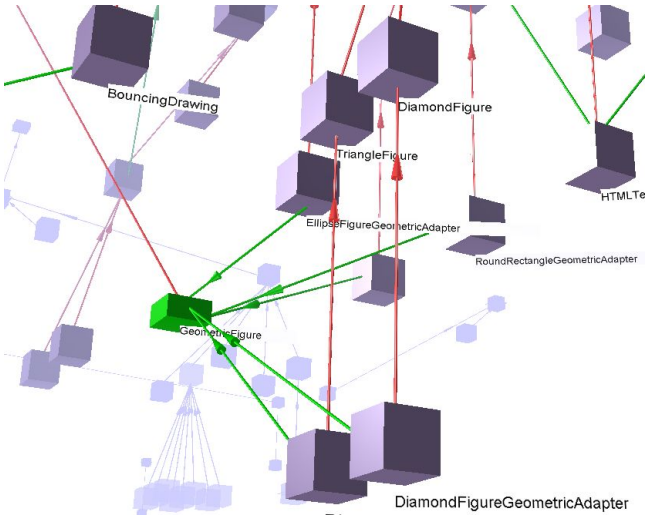


Figure 4: Class Hierarchy

Particularly in a three-dimensional setting, tool support is mandatory and must encompass both interaction with and navigation in a 3D-scene. We have implemented such a tool supporting the above-mentioned diagrams. It is briefly described next.

## 2 Tool Description

Our tool VisMOOS (Visualisation Methods for Object-Oriented Software Systems) is tailored for visualising code structures in Java software systems. VisMOOS uses so-called visualisation methods to selectively display different aspects of source code with different geometric arrangements taking into account criteria for good visual perception. The tool, firstly, analyses a Java source code using JavaDoc, and, secondly, generates a structural model. Thereupon, the user selects a visualisation method to view a 3D-scene rendering the structural model using the Java3D technology and spring embedding algorithms. Each method offers interaction techniques – such as rotation, zoom, fade, elision, or setting a degree of interest – by which the user can interactively explore the scene to gather information he or she assumes relevant for maintaining the software under consideration. As VisMOOS is a plug-in for Eclipse, it can be utilised within programmers daily work and integrates into any software development process. Particularly for large software systems, VisMOOS suitably offers a flexible filtering mechanism by which the user can reduce the amount of data displayed or focus on the most essential components he is interested in. In addition, we consider it helpful to support the user in exploring a scene by means of user-defined metrics. That is, the tool will allow to point on particularly interesting situations using a data-mining approach for graphs (cf. [Cook and Holder 2000]). So far, we have implemented

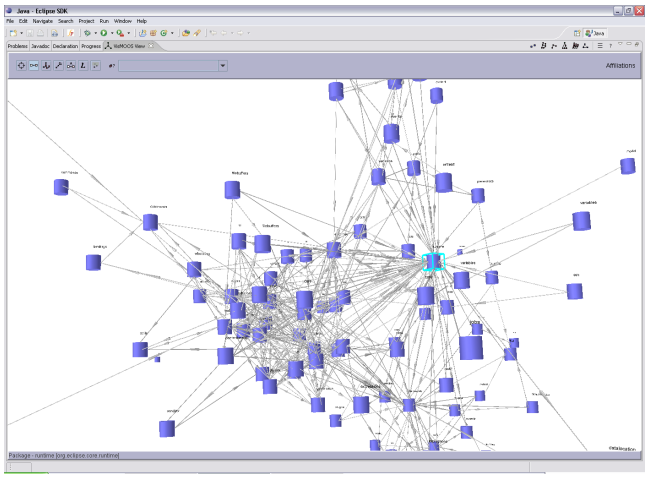

Figure 5: Eclipse package affiliation relations

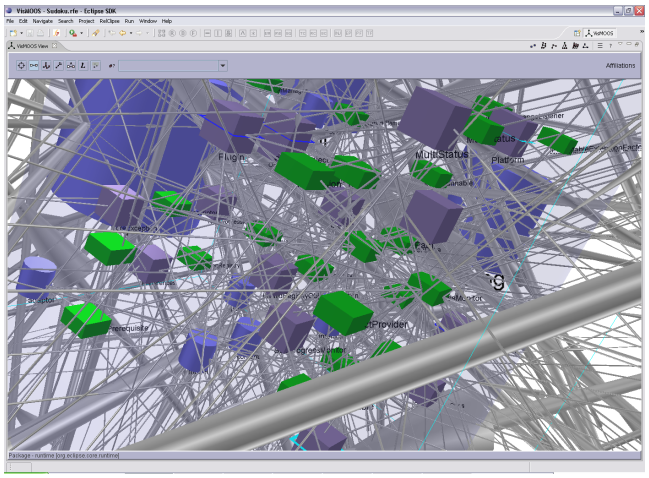

Figure 6: `eclipse.core.runtime` package network

different metrics to quantitatively analyse code structures, though visualising the analysis results within the scene is currently not implemented. To explain how to explore a software system with VisMOOS, we briefly introduce a case study and discuss some of the aspects that can be discovered in our relation diagrams. The tool can be downloaded at `http://ls10-www.cs.uni-dortmund.de/vise3d`.

We are already using VisMOOS in an industrial setting and can report on positive feedback. With empirical studies, however, we want to discover why the concepts used in VisMOOS are effective, and what exactly are the reasons for their success. That is, we expect an analytical consolidation which a case study alone might not adduce. To give a first impression of relation diagrams in use, we briefly discuss a case study before we propose empirical experiments to analyse cognitive requirements for software maintenance supported by our reverse engineering tool VisMOOS.

## 3 A Case Study in Code Structure Visualisation: The Eclipse Source Code

Recently, we have explored several software projects of different size: A student project encompassing 26 packages and 120 classes, an industrial software project encompassing more than 800,000 lines of code arranged in several hundred packages, classes, and interfaces, and the Open Source Project 'Eclipse' (without considering the Java Development Toolkit (JDT)) encompassing over 500 packages, 7200 classes, and 1400 interfaces.

In [Fronk et al. 2006], we reported on the Eclipse source code to which we refer here to explain how to explore unknown large software systems. We follow a top-down approach, i.e., we first visualise the dependencies between the Eclipse packages showing affiliated classes and interfaces in an integrated and hierarchically organised package affiliation diagram (the visualisations printed here and more useful when viewed within VisMOOS and only serve as a qualitative impression of the complexity the Eclipse source code possesses). From the layout of the scene in Fig. 5 one can deduce that the packages `eclipse.ui` and `eclipse.core.runtime` are the most essential ones: The classes they contain possess the largest number of associations to classes in other packages. Most interestingly, the classes from these two packages mainly communicate with each other via classes found in the package `ui.internal` which is the package with the largest number of lines of code (LOC), viz 49898. The packages `osgi.framework` and `internal.core` also form very important communication nodes, although their size is relatively small: 5924 LOC and 2760 LOC, respectively. From metrics alone, such a finding could not have been made. A process automatically searching for components that satisfy such constraints is under work.

In a second step, one may explore the interior of packages one is interested in. For instance, the sub-packages, classes, and interfaces found in the `ecplise.core.runtime`-package (see Fig. 6) form a very dense association network. To analyse it from scratch, i.e., without domain-knowledge about which packages or classes are intended to be used for which purposes, we consider quantitative properties first.

It is easy to see from the layout again that there are some classes and interfaces playing a central role, and applying a suitable filter shows their associations are mostly leading to classes located within a large variety of packages and outside of the `ecplise.core.runtime`-package. Consequently, it is a third step to analyse associations omitting the package affiliation context in order to concentrate either on method call graphs and thus focus on dynamic properties, or on code organisation by means of relations between classes and interfaces and thus focus on static properties. It is the latter that we assume more important when understanding code structures is on center stage.

Although the class `ui.texteditor.AbstractTextEditor` with 4096 LOC is the largest class in the `eclipse.ui`-package, it does not constitute the most interesting association network with respect to code organisation. Moreover, running some metrics on the code identifies the class `jface.text.TextViewer` to implement 11 interfaces, which is the largest number of interfaces implemented by a class. A look at its association network (Fig. 7) with `TextViewer` positioned in the center offers a very nice star-like organisation to the interfaces implemented and the classes used. The latter are almost totally unrelated and implement further interfaces.

Sticking to interfaces, metrics show that the interface `jface.viewers.IStructuredContentProvider` is implemented 83 times, followed by `core.runtime.IAdaptable` (55 times) and `jface.util.IPropertyChangeListener` (53 times). In a forth step, it is interesting to see how class

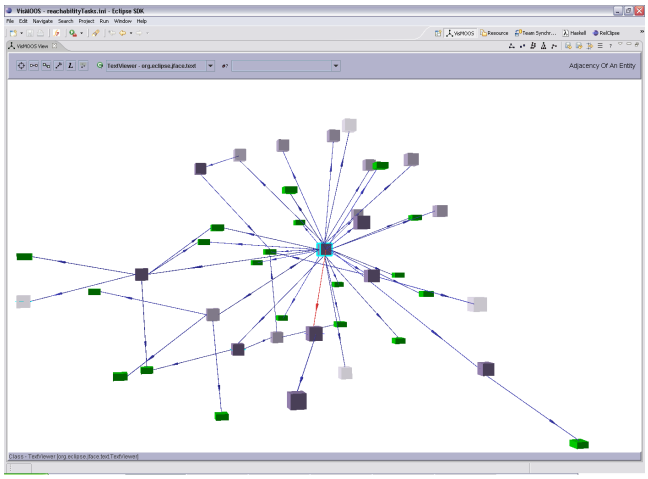

Figure 7: `TextViewer` association network

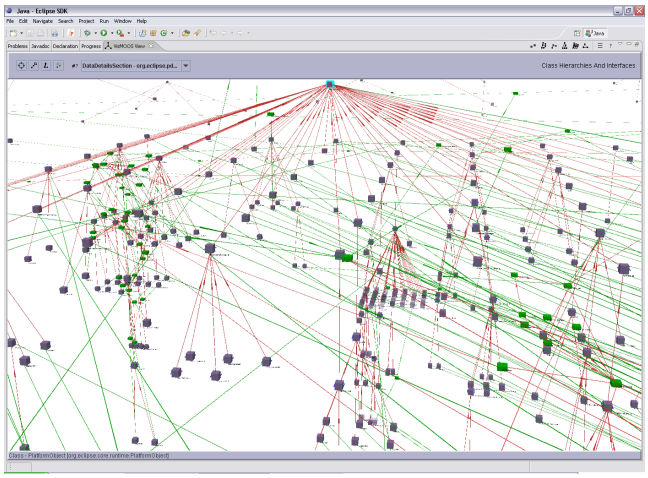

Figure 8: A sample Eclipse class hierarchy

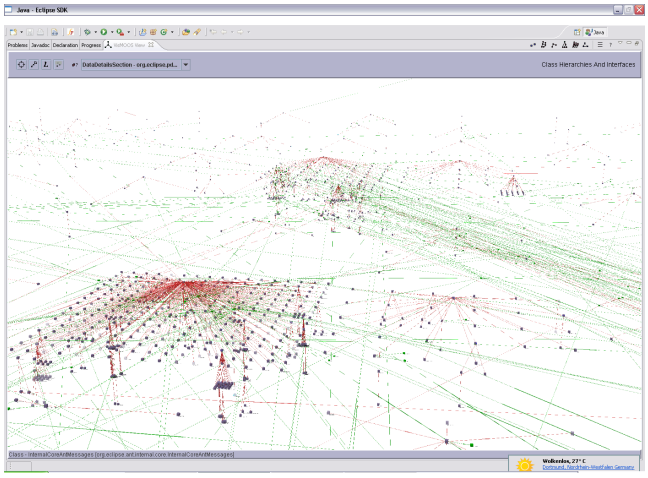

Figure 9: Small portion of all Eclipse class hierarchies

hierarchies are organised and how interface implementations spread over them. From metrics we learn that the class `jface.action.Action` has 275 subclasses, followed by `arg.util.NLS` and `jface.viewers.LabelProvider` with 113 and 110 subclasses, resp. These figures allow to estimate whether there are a few large hierarchies or many small one, yet they say nothing about their hight, i.e., whether class hierarchies are flat or rather deep. In Eclipse, there are many medium-sized hierarchies, and only a few classes have a very large number of subclasses the organisation of which is nice to see in class hierarchy diagrams (see, e.g., Fig. 8).

From a class hierarchy diagram displaying all class hierarchies available, we learn that the classes `jface.action.Action`, `jface.dialogs.Dialog`, and `core.runtime.PlatformObject` constitute the most central hierarchies. Furthermore, since interface implementation relations are also captured, we can see that the `PlatformObject` hierarchy in particular contains many classes implementing the interface `IAdaptable`, a fact that can easily be deduced from layout again: Fig. 9 shows a lot of green lines representing interface implementation relations leading from the above-mentioned class hierarchy to `IAdaptable`.

## 4 Facets of Visualisation Evaluation

Software comprehension demands to quickly and precisely overview relations between software entities. Their visualisation must allow to deduce the information needed for maintenance. Hence, visualisations can be evaluated at least considering the following facets embracing syntax and semantics:

**Concrete Syntax:** The way data are represented should coincide with what the data mean. That is, a linked list, for instance, should be represented as such in order to allow to find structural or logical bugs quickly and effectively, or different colors should be used to mark different types of data.

**Interpretability:** Depending on the specific task, different data are needed. For example, debugging requires different data than refactoring. A visualisation should thus be *task-specific* and *completely* offer the data needed, if available, which in turn should be visualised in such a way they can be interpreted *correctly* with respect to the users intention.

Since we use different geometric arrangements for different kinds of relations between code entities, integrate them with each other, and use the third dimension simply for having more space and layout possibilities available, we have the strong feeling that compared to UML diagrams our relation diagrams support a better visual ingress on the code structures displayed and enhance code comprehension for maintenance purposes. Nonetheless, we offer at least new insight into code structures through presenting aggregated information on large software systems. With this opinion and from the above-mentioned facets we can deduce some assumptions to be tested on our 3D relation diagrams when compared to the corresponding UML package and class diagrams:

- They allow to memorise more structural aspects better.
- They allow to recognise ni less time more code changes in the underlying source code carried out by other programmers.

- They allow to find and recover specific structures more easily and more accurately.
- They allow to qualitatively assess code structures more accurately.
- They allow to quantitatively assess code structures more precisely.

We are planning to conduct a two-step evaluation process (cf. [Dumas and Redish 1999]). In the first step, the assumptions will be evaluated within a preliminary survey to check whether or not they hold. Simultaneously, we expect to obtain indices about what proves them true or wrong. That is, the first step will also be a hypotheses-generating study helping to formulate precise hypotheses such as, e.g.:

- The larger the underlying source code, the more 3D relation diagrams are superior to UML diagrams.
- Colors support structure memorisation.
- Layout is responsible for structure recognition.
- 3D-space enhances overview on large sets of data.
- The features offered by VisMOOS support code assessment.

In a second step, continuative experiments will empirically test such and other hypotheses to definitively discover the reasons why and not or where and where not 3D relation diagrams work. Currently, we are designing the first step on which we report in the remainder of this paper.

## 5 Preliminary Study: Design and Conduction

We are currently recruiting students from both Bachelor and Master Programs of Computer Science at the University of Dortmund. They possess different skills in reading UML diagrams and none in reading 3D relation diagrams. Both groups will encompass about 10 students, i.e., there will be about 20 students available. Before we let them participate the experiments, a short briefing will introduce them into reading both UML and 3D relation diagrams. Then, the students are arbitrarily divided into two groups. In each experiment, one of the groups will deal with a UML diagram and the other with a 3D relation diagram to fulfill a given task. The diagram assignment will be switched from time to time such that each student will face either diagram type. Both groups are tool supported: the UML group will use OMONDO, and the 3D group will use VisMOOS, such that both groups rely on the Eclipse environment. The groups are not allowed to use the search-feature of either tool but to navigate through the diagrams manually.

We are planning to conduct the following experiments:

- A complex diagram is given to the students. They are asked to look at it for 10 minutes and to gather as much 'knowledge' as possible. We do not tell them on what they should concentrate. After this phase, there will be a 20 minutes break in which the students are shown some part of a relaxing movie. After that, they are asked to write down what they remember. We expect the UML group to remember technical details in particular, whereas the 3D group should memorise structural conspicuities. With this experiment, we want to assess the quality of our visualisation w.r.t. productivity through commemoration aid.
- Each group is given a complex diagram. They are asked to look at it for 10 minutes. Then, the groups are given a set of diagrams showing similar situations. However, only one diagram is equivalent to the first one except for some subtle changes in the relations between the entities shown. The task is to find this diagram. This experiment aims at structure recognition. We expect the 3D group to appoint the correct diagram in less time with a smaller error rate.
- Each group is given a complex diagram. They are asked to look at it for 10 minutes. Then, each group is given a list of entities (packages, classes, and interfaces) and asked to describe the relations between them within 5 minutes times. This experiment aims at orientation within the code. We expect the 3D group to give more accurate answers in less time.
- Each group is given a class diagram. The task is to assess within 5 minutes time the effect the deletion of a certain method from a specific class will have on the the source code visualised. This experiment aims at assessing the relevance of classes in their association context.
- Each group is given a set of three times three diagrams showing three different views on three different projects. The task is to collate within 5 minutes the three views belonging to the same project. This experiment aims at evaluating the qualitative assessment of code structures. We expect the UML group not to pass this test if package and class names are deleted from the diagrams, whereas the 3D group is then still able to fulfill the task. In case the names are not deleted from the diagram, the UML group may finish this task slightly faster.
- Each group is asked to add a new feature into the product visualised through a class diagram containing classes, abstract classes, and interfaces. The task is to assess within 3 minutes the cost of this code change by estimating the number of classes and interfaces affected by the change and to propose a change strategy (which class to adapt first to what, and so on). We expect the 3D group to give more precise answers and propose better change strategies.

From these experiments we are expecting initial empirical hints on the quality and effectiveness of our approach addressing the enhancement of code comprehension for maintenance purposes, and whether VisMOOS is supposed to be a useful reverse engineering tool. Nonetheless, since we already use the tool in an industrial setting, we can already report on positive feedback and hope to corroborate it in this ongoing study.

## References


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

Cook, D. J., and Holder, L. 2000. Graph-based data mining. *Intelligent Systems and Their Applications 15*, 2 (Mar.), 32–41.

Dumas, J., and Redish, J. C. 1999. *A Practical Guide to Usability Testing*. Intellect Books.

Fronk, A., Bruckhoff, A., and Kern, M. 2006. 3d visualisation of code structures in java software systems. In *Proceedings of the ACM Symposium on Software Visualisation*, ACM SIGGRAPH, 145–146.

Lano, K., and Haughton, H. 1993. *Reverse Engineering and Software Maintenance: A Practical Approach*. McGraw-Hill, Inc.

Mayrhauser, A., and Vans, A. M. 1993. From code understanding needs to reverse engineering toolcapabilities. In *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering*, 230–239.

Rekimoto, J., and Green, M. 1993. The Information Cube: Using Transparency in 3D Information Visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems*, 125 – 132.

Rohr, O. 2004. *Auswahl und Konstruktion von dreidimensionalen Visualisierungsmethoden zur Exploration objektorientierter Softwaresysteme*. Master's thesis, Lehrstuhl Software-Technologie, Universität Dortmund, Germany. In German.