Copyright 2001 IEEE.

Published in the International Conference on Program Comprehension (IWPC)

May 12-13, 2001 in Toronto, Canada.

Personal use of this material is permitted. However permission to reprint / republish this material for advertising or promotional purposed or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

Contact
Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / PO Box 1331 / Piscataway, NJ 08855-1331, USA.
Telephone: + Intl. 908-562-3966.

Mediating Diverse Visualisations for Comprehension

Claire Knight and Malcolm Munro
Visualisation Research Group,
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
{C.R.Knight, Malcolm.Munro}@durham.ac.uk

Keywords: Software Visualisation, System Visualisation, Component Brokerage, Usability

Abstract

For ease of use, comprehension, and general acceptability it is important that tools that purport to aid comprehension are able to provide a variety of customisable views. The most important underlying factor is the wide differences between users in terms of their culture, experience, visual together, working preferences, and general abilities. Comprehension studies have also shown that different views, and the ability to selectively filter data are important. Combining all of these ideals into one, automated, infrastructure is the focus of this paper. Component technologies provide a way of making this a reality, and also for providing visualisation creators with a general application into which they can leverage their work. Advances in this arena have allowed for component brokerage and automatic dynamic data connections to become an actuality.

1. Introduction

It has long been acknowledged that to successfully promote comprehension of any dataset, it is important to provide different ways of viewing that data, and allowing choices to be made about the data being displayed. There is now also much to support the view that personalisable interfaces and customisation of the information presented is important for not only acceptance, but also usability and comprehension.

Program comprehension, and indeed many other avenues of software engineering, have made much use of diagrammatic representations. Unfortunately, despite the changing nature of software in terms of size, complexity, languages, distribution, etc. the representation style has remained constant. Whilst familiarity of any form of interface or representation provides a comforting first contact for the user, newer users or those whose needs dictate that more adequate displays are used are still not catered for.

There is the luxury within software engineering and it's related disciplines in that practitioners (at least theoretically) know about the many tools and techniques that are available to them. This awareness and exposure has lead to the development of the tool presented in this paper for the purposes of visualisation and program comprehension. Much of the process of comprehension is goal oriented, and specific goals will have dedicated information requirements. These in turn may drive the sorts of views that are (a) necessary and (b) appropriate. The aim of this work is to provide a visualisation arena containing a variety of visualisations, judged as appropriate from a visualisation perspective, that can be linked to various data sources as required by the user. This work is novel in that it seeks to automate this process through data brokerage, and to provide a possibly distributed, customisable, and "pluggable" environment in which such visualisations can be used. It does not seek to further component research, but to utilise the techniques that have come from this area. Live data links can be utilised, themselves possibly as remote as the various visualisations. This allows for great flexibility in the pursuit of understanding, and also paves the way for such tools and sub-tools to be provided on a service/as-needed basis, as indeed is the way in which software delivery is seen to be heading.

2. Visualisations

Software visualisation can be seen as a specialised subset of information visualisation. This is because information visualisation is the process of creating a graphical representation of abstract, generally non-numerical, data. This is exactly what is required when trying to visualise software. The term software visualisation has many meanings depending on the author. For the purposes of this paper software visualisation can be taken to mean any form of program visualisation that is used after the software has been written as an aid to understanding (i.e. it does not mean visual programming). More formally software visualisation can be defined as [1]:

> *"Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration."*

This section on visualisations starts by addressing software visualisation over the last ten years. It then briefly addresses the controversial issue of evaluation. These apparently diverse sections are provided as background to the work presented later in the paper as they contain issues that are important in a wider context.

2.1 Program Comprehension Visualisations

The examples in the following four sections are representative of the advances made in software visualisation in the last ten years. The first three are refinements and enhancements of node and arc technology with improved browsing and node management, and the final one is one of the early software visualisations to move away from total reliance on nodes and arcs. Whilst these four tools show slightly different aspects of the program code, they are all intended for the tasks of program comprehension and software maintenance hence their inclusion.

2.1.1 SHriMP

SHriMP (**S**imple **H**iera**r**ch**i**cal **M**ulti-**P**erspective) [2] is a visualisation that combines the nodes and arcs representation so cherished by software visualisers with the fisheye filtering technique. As the name implies, the fisheye technique emulates the behaviour of a fisheye lens. The information at the centre of the view is magnified, whilst that at the periphery is reduced in size. SHriMP also incorporates the use of nested graphs for the display of software structures. The use of these two techniques provides, according to the authors, the ability to create multiple views at different levels of abstraction and perspective. To be able to interact with the data in this way is a powerful way of dealing with large amounts of information as selective filtering and viewing can take place. Such extensions are a step towards being better able to deal with the large and complex software that is so pervasive today, but the user of two-dimensions and the standard representation as the basis may limit the applicability and use of such tools.

2.1.2 VIFOR

VIFOR stands for **V**isual **I**nteractive **FOR**tran and is a software tool that is geared towards maintaining Fortran 77 code [3]. This system works by using a database of code detail from the Fortran code and then allowing it to be viewed and queried in either the textual form of the code or a graph layout based visualisation. This layout mechanism was developed for the VIFOR tool and attempts to merge the standard call graph and data dependency graphs that are more commonly used.

Early work on a C maintenance and understanding tool is also documented in this paper; VIC. An extension of this work was the development of VIFOR 2 [4]. This improved the browsing system to allow the integration of incremental recording and retrieval of documentation.

2.1.3 CARE

CARE (**C**omputer **A**ided **R**e-**E**ngineering) is an understanding tool that works with C source code [5]. This understanding tool makes use of two-dimensional visualisations in windows and browsers (as with the previous tools) to show graphs of some of the code relations. As with VIFOR, CARE displays the data flow and call structure of the program using an extension of the VIFOR layout algorithm. In order to do this, a repository of the structural and functional dependencies in the code is generated, and the presentation part of the tool uses this information when displaying the visualisations. The tool also supports the creation and use of both graphical and textual slices through the information.

As with VIFOR, an extension of the tool was developed to deal with other languages. OO!CARE [6] is an extension of CARE that deals with C++ code, hence the **O**bject **O**riented addition to the name. It is also able to deal with C code because of the syntactical similarities of the language notwithstanding the object oriented part of C++.

2.1.4 SeeSys and SeeSoft

In an attempt to address some of the shortcomings of relying solely on nodes and arcs for data representation, the tools SeeSys [7] and SeeSoft [8] were developed. These tools are part of a research effort that produced similar displays for several underlying data types. The visualisation technique used by these systems is based on the idea of decomposition of the information to be visualised into its component form. Colour and interaction are incorporated into the systems, and the displays make much use of colour scales to visualise extra information about the underlying data. The system also uses the overlay of additional information onto the display to provide yet more facts for the user.

SeeSys is a visualisation system for software metrics whilst SeeSoft visualises the program code and the constituent files. These visualisations are based on three principles:

1. The individual components can be assembled to form the whole. This allows the user to easily see the relationships between them.
2. Pairs of components can be compared to understand how they differ.
3. The components can be disassembled into smaller components. This important feature of the components allows the structure of the display to reflect the structure of the software.

The individual components are visible whilst maintaining a view of the whole system. This sort of technique has also been applied in the Information Mural visualisations of Jerding and Stasko [9], whilst the concept of encoding wear as a coloured property of source code was originally documented by Hill and Hollan [10].

2.1.5 Summary

The previous four visualisations present only the more traditional views. They do not cover some of the more adventurous new approaches such as FileVis [11], Software World [1], or the visualisation of software system architecture for comprehension [12]. Such approaches are considered to be very worthwhile but their acceptance is not universal because of the difference in evaluation of such techniques. The following section goes on to cover some of these issues, but suffice to say these would all be just as suitable candidates for encapsulation as visualisation objects for integration into a mediated visualisation system.

2.2 Visualisation Evaluation

Concurrent work in progress by the authors is related to the need to provide evaluations of the effectiveness of visualisations. Currently there are influences from program comprehension, virtual reality (for 3D visualisations), usability engineering, human-computer interaction, and to a small extent information visualisation. There is a problem in that system and software visualisations are often seen as being all things to all people! In reality several factors from each of these areas can be considered important.

An early conclusion is that the best way of expressing the suitability or effectiveness of a visualisation is:

Effectiveness = suitability for task(s) + suitability of representation, metaphor, and mapping based on the underlying data.

A simple example of where current work [15] falls short is that the display is considered to only be 2D, thus many of the issues surrounding 3D navigation and orientation are not addressed, and the answers to the existing criteria could be misleading if the number of graphical dimensions are not taken into consideration. Another is through the concept of usability. Many usability studies do consider the task the user is trying to achieve, but do not cover the possibility of investigative type tasks that are common when carrying out comprehension. The user may jump around the data, revisit areas, and the ultimate success of the task may actually be that something is not there; such as there is no direct impact of a change.

This is a broad definition and the content of the two extensive categories is under constant refinement during the process of work described further in this paper. The tool that is the focus of the paper then also provides a suitable test-arena for examining the detail within such evaluations and claims.

3. Components

Component oriented software is now the development of choice because of the flexibility that it allows for adding and removing sections of systems, reuse, and maintenance. Whilst component technologies can run the risk that many encountered with object oriented languages (they are not silver bullets!) they provide several new avenues along which the development and maintenance processes can travel.

Distributed components (utilising techniques such as remote invocation) and more recently the concept of system delivery and negotiation have been created in response to the changing nature of both software delivery and the speed with which businesses need to adapt to market change. The use of the internet, and a much more global focus to many business activities furthered this development, and again provides another rich vein for software engineering,

This section highlights some aspects of Java (the implementation language for this research) that are, or can be, utilised to encourage a flexible system. The areas in which these features can then be expanded, and used more fully, are discussed. This is because the ultimate aim of a tool such as this one, as well as aiming to address usability issues and allowing comprehension to take place using sub-tools of the users choice, is to have a flexible environment into which developers can make available more tools as techniques and knowledge develops.

3.1 Features of Java

There are many features of Java that can be used in a tool such as this one. The most obvious is that through the nature of the language, any tool produced can be used on several platforms; immediately contributing to the usability aspect.

The true object oriented nature of Java (rather than C++ which allows floating C code to exist) allows for containership of the broker tool and the various visualisation elements that may be used with it. Java obviously has dynamic binding (as it is object oriented), but it also has dynamic class loading, which can be done as other code is running. This allows object references to visualisations (in this case) to be selected by the user and then integrated into the system on the fly.

Another useful aspect of Java is the ability it provides for creating graphics by painting to specific components. There is a well-defined graphics system so that the contents can be concentrated on, and then the virtual machine deals with the lower level, platform and system specific detail. Such graphics, or even the visualisations, can the implemented in threads so that multiple visualisations can monitor live data sources and update themselves as necessary without the user having to relinquish control of the machine or the parent broker application.

The event driven model of the interface APIs also allows for efficient control of message passing and monitoring other classes. Changes in data, for example, can be notified to only those other objects that have registered an interest in knowing about such changes.

Newer features of Java, such as JavaBeans, allow for classes to be known to adhere to certain formats regarding properties, events, and public interfaces. This allows for more efficient reuse and automation. Extending this concept allows for all add-ins to the broker to have to implement certain pieces of code that the broker then knows it can rely on.

3.2 Expansion

Enhancements to the JavaBean model have led to EJBs (Enterprise JavaBeans) in which business logic can be separated from the “doing” code. This could lead to the linking of objects that are actually running parts of other systems. Comprehension can be carried out on the fly as the system continues to work in the vein in which it was intended.

Remote method invocation (related to CORBA concepts) goes some way towards allowing for the fully distributed system to exist. Introspection can also be used on any running class to examine it’s contents and monitor what it is doing during runtime. Object access methods, and even properties of that object can be determined automatically from running objects and this information utilised for integration of (remote) visualisation objects.

Serialisation of running objects allows for them to be converted to byte streams and saved for later use. These byte streams can then be reassembled and run at another time. Adding this facility to objects in a visualisation tools allows for views and arrangements of various visualisation and data links to be saved mid-use and then returned to. To implement this requires that all plug-in pieces of code themselves implement such functionality, but this is not beyond the bounds of future advancements.

4. Brokering Interface

This section introduces the visualisation broker interface that has been referred to in the previous parts of this paper. Much of the rationale for creating such a tool has been covered in section two, and many of the pertinent Java features introduced in section three. For more detail on Java see [14].

The broker makes use of a multiple window interface, conventionally referred to as MDI (Multiple Document

Interface) in Microsoft Foundation Class programming. This allows for a parent application to create and manage many sub windows, which in this case are used to hold the visualisation or data objects. By using this model, a user of the tool is then able to link the visualisations and data sources that they require from a task perspective and a personal preferences perspective. They can then use multiple views (in different display types and metaphor if required) to examine the data and provide themselves with enough information to set about solving the task in hand.

4.1 Achievements

There are several aspects of the broker comprehension tool that are highlighted in this section. These have been chosen to illustrate how this research builds on that done by Storey [15] on program comprehension tools, guidelines from Von Mayrhauser [16] amongst others.

The broker application main window is as shown in Figure 1. The broker is very simple when no plugins have been registered and used within it, and provides only a window front end to the application with menus allowing for plugins to be used with the application. This has been done so that the application is customisable from the start, and that certain defaults do not exist to give the impression that these are the only possible types of data and ways of looking at them graphically.

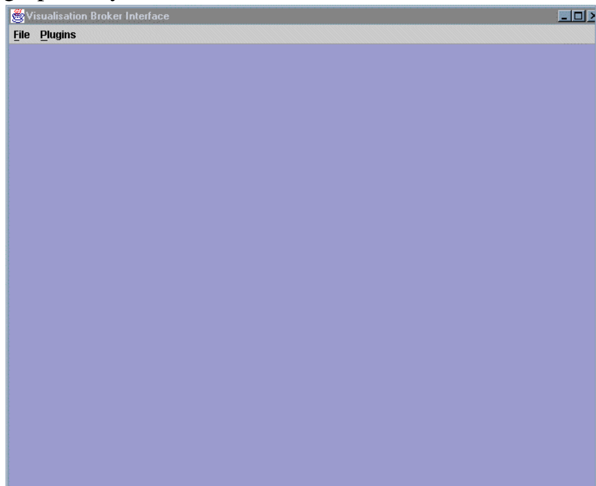

Figure 1 - Main application window

By providing what is effectively an empty window, the broker application provides a way to contain any of the plugins subsequently registered and loaded. This approach supports the use of internal windows so that all analysis is contained and not cluttering up the users desktop. It also allows for windows to be brought to the top together and events to be sensibly handled, an example being a visualisation entity that has a control panel, and therefore requires two windows.

Because of the plugin approach adopted for this research, and theoretically moving towards using mediation and brokering between components many different visualisations can be used. These can differ in the metaphors and representations used, the number of dimensions employed for the graphics, and the type of data that they are suited for. Many different sources of data can also be integrated, either with the same visualisation style or each with its own suitable graphical display.

An instant benefit of using other data sources and visualisations within an umbrella environment is the ability it gives to the user to be able to annotate their findings and knowledge, and to link this across views and data sources. Often this has only been supported within each visualisation (if at all), and whilst specific annotations directly onto the graphics may be desirable much of the time it is not. An area of future work may indeed be to interact directly with the visualisation in order to support this as well. This would also lead towards successfully creating linked windows, so that for any data source item, it could be located in any of the registered graphics currently displaying that data.

In summary, it can be said that the broker tool is customisable, even at this basic stage, is able to support multiple views, even of multiple data sets, provides the potential for cross browser (from a visualisation perspective) annotation, reference, and even linking of actions, and finally that it is interactive in both the visualisations and their control, and the broker tool.

4.2 Demonstration of features

This section has been included so that the use of the broker can be seen. The visualisations provided, and also the degree of control over them (rather than at the plugin level) are generally of a basic nature. It is appreciated that much more sophisticated visualisations exist and would be better suited for complex analysis. They have not been included for the simple reason that this demonstration intends to show the feasibility of such an approach and that the level of graphics is, for this purpose, a secondary issue.

This demonstration will show how data and visual sources can be registered with the broker application, and then how they can be linked. At the moment this is

done at a rudimentary level, although future enhancements are to include more detailed data descriptions for which parsers can be generated on the fly and then even integrated with database accesses (such as through JDBC). Use of the control panels of the visualisations, and dynamic data change will also be shown. This is to highlight the possibilities afforded by this research of fully dynamic, remote, investigation. It is appreciated that the data shown in these examples is only repreentative of, for example, that which would be found in a call graph. The aim is to show the principles of this work, not to perform some detailed analysis of the graph.

From the main window (Figure 1), there are two possible menu options. The second of these is *Plugins* and it is what will be used for this demonstration. The possible options on this menu can be seen in Figure 2.

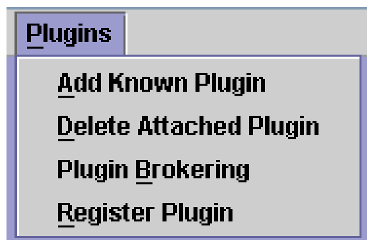

Figure 2 - Plugin Menu; options available

From this all the necessary loaded and linked of plugin visualisations can be carried out. This process will be shown through the use of two auxiliary applications.

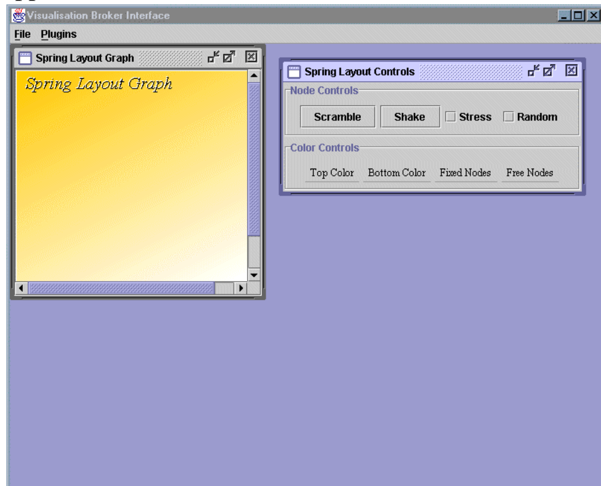

Figure 3 - *GraphVisBean* plugin in the broker application

The first is a Visualisation application that draws graphs of any given data based on node names and the connecting edges. The second is a Data only application that maintains a data set that can be used by other applications, but also allows a user to modify that data on the fly. Figure 3 shows the visualisation broker after the *GraphVisBean* has been added. As can be seen there is no graph showing in the visualisation panel because there is currently no data to drawn. The window to the right shows a control panel that allows colour changes of the visualisation, and also some configuration options to be set for the graph itself.

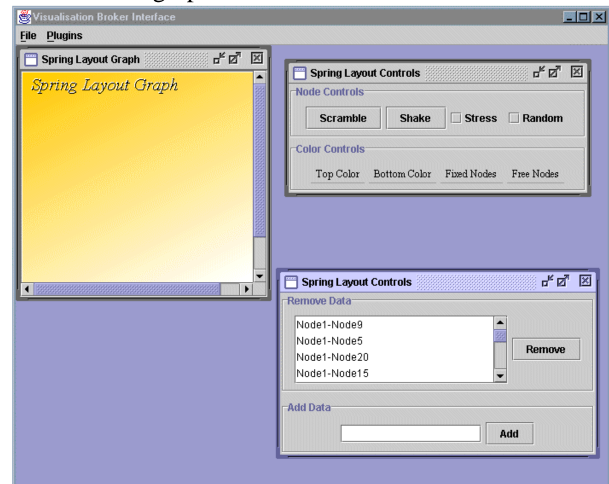

Figure 4 - *DataListBean* added to the application

The next thing to add is the *DataListBean* so that some form of visualisation can actually be displayed by the visualisation. Figure 4 shows the application after this bean has been loaded. Please note that there is still no graph drawn. This is because no link has been created between the two. The control panel of the data can be seen with a list of current connections in the graph data, and providing the facility for adding and removing connections.

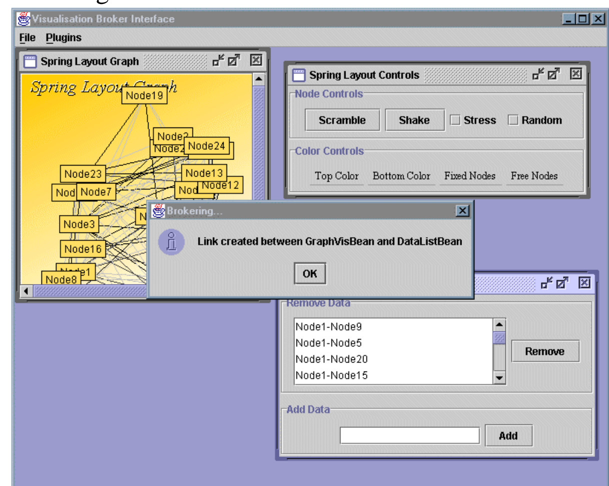

Figure 5 - Linked plugins in the broker application

Figure 5 shows the result of linking the two. The graph is now being displayed along with a message to

the user confirming the link that has been created between the two.

Whilst spring layout algorithms are often in a slight state of flux, a fairly consistent layout usually emerges as an application is running. To see how live data changes affect the plugins, Figure 6 shows the result of removing a connecting edge. The graph structure has changed.

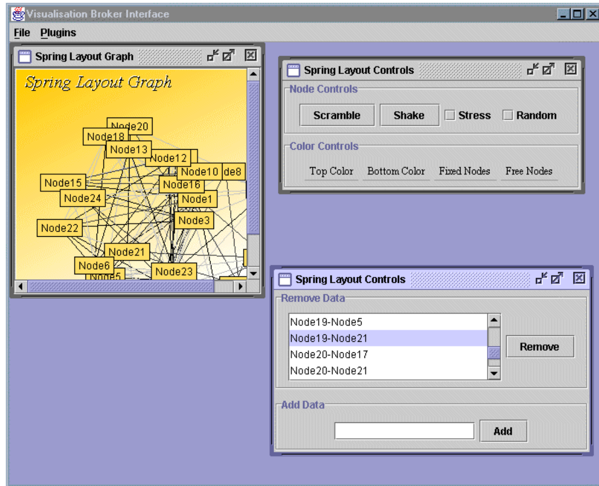

Figure 6 - Changes due to dynamic data updates

5. Further Developments

There are many further developments relating to this work. Many of them relate to visualisation research questions, and this tool provides a testbed in which to examine them. There are also many at the level of enhancing the capabilities of the broker tool itself, many of which can be seen from the information provided in the Java section (three) of this paper. Finally there are broker applications whereby the visualisation and broker enhancements are addressed, and the use of such techniques can then be used to improve the application.

5.1 Visualisation

The first step for visualisation advances, is to create visualisations of the same nature as Software World [1] or FileVis [11] that can be utlised as plugins. The main reason for doing this is that it is then possible to show how different kinds of views can be of use alongside the more traditional views. If both can be used side-by-side with the same data, the concept of personal preference can be ignored to show how answers can be achieved from both. Another way of showing how these visualisations are of use is to make them available in this way so that those who prefer that form of display and interaction are free to do so.

By furthering the question of visualisation evaluation in parallel to this work, evaluations that do not confuse or cloud the issues surrounding 2D and 3D images can be developed. This will be helped through the development and user testing of a variety of visualisations, even those that may be considered to be too specific for use as a visualisation tool. This latter area of small, specific, visualisations, is also one that is useful in the context of the broker. Visualisations of this form may only have application at certain points during the process of knowledge discovery and refinement. Normally they would be discarded as visualisations for not scaling, or for falling short in other areas. The use of a tool such as the broker allows for them to be tried and if successful, incorporated as necessary by the user.

5.2 Broker Applications

Considering areas that are wider than program comprehension, or that have very directed needs, broker applications may be created through restricting plugins, and then distributing such tools (the broker application and a subset of plugins) for specific tasks. Whilst this limits the customability, it allows for specific aims to be addressed and easily encompassed into usable knowledge discovery and management tools.

The concept of true mediation and brokerage between plugins by the application allows for much more sophistication. This is an area that is also being researched from other component perspectives and much may be learnt from elsewhere. What is important for development in this case, is being able to deal with the specific needs of program comprehension data sources. Information visualisation applications may well have their own specific issues but they are currently beyond the scope of discussion. Issues such as parsing static code files, linking with dynamic analysis tools, and even with running code (itself under analysis) creates a set of problems that are very specific to this field. There are solutions to some of these, but creating extensible, component oriented, automated visualisation aware solutions is something that requires more work.

5.3 Summary

There are many unanswered questions that have emerged as part of this research concerning the nature of component software utilised in this manner. Some are a matter of design decisions and can be progressed much more easily. Others are more complicated, such as effective ways of registering unknown and remote visualisation entitities and mediating between different

data demands. The one thing that is clear however, is that there is plenty of potential from this research, much of which is also equally applicable beyond program comprehension.

6. Conclusions

This paper has demonstrated how developments in software engineering component technologies can be utilised to solve existing problems within software engineering. In this case the focus is program comprehension through visualisation. Both of these are also important research areas and the rationale for this work is that because of the subjective nature of visualisation tools, it is vital to allow for personalisation. Such tools also provide a suitable forum for the testing of evaluation theories, and usability issues associated with particular visualisations.

The important features of the research and the tool have been presented, as have several areas in which it is felt important to enhance and further this work. The research has also been shown to permit identified good features of program comprehension tools such as multiple views and interactive browsing. It is therefore a strong foundation for further additions and refinements. In conclusion, this paper has shown that through the adoption of different techniques such as component usage, the problems of having to successfully visualise and comprehend the ever increasing size and complexity of systems and software can be a realisable goal.

Acknowledgements

This work is financed by an EPSRC ROPA grant: VVSRE; **V**isualising **S**oftware in **V**irtual **R**eality **E**nvironments. The spring layout algorithm and some of the node drawing code is based on demonstration code provided by Sun with the Java Development Kit, although changes have had to be made for the swing graphics and dynamic nature of *GraphVisBean*.

References

[1] C. Knight and M.Munro, *Comprehension with[in] Virtual Environment Visualisations*, Proceedings of the IEEE 7th International Workshop on Program Comprehension, pp4-11, May 5-7, 1999.

[2] M-A. D. Storey and H. A. Müller, *Manipulating and Documenting Software Structures Using SHriMP Views*, Proceedings of ICSM '95, pp 275-284, October 17 - 20, 1995.

[3] V. Rajlich, N. Damaskinos, and P. Linos, *VIFOR: A Tool for Software Maintenance*, Software Practice and Experience, Vol. 20, No. 1, pp67-77, January 1990.

[4] V. Rajlich and S. R. Adnapally, *VIFOR 2: A Tool for Browsing and Documentation*, IEEE International Conference of Software Maintenance, pp 296-300, 1996.

[5] P. Linos, P. Aubet, . Dumas, Y. Helleboid, P. Lejeune, and P. Tulula, *Facilitating the Comprehension of C Programs: An Experimental Study*, Proceedings of the 2nd IEEE Workshop on Program Comprehension, pp55-63, July 8-9, 1993.

[6] P. K. Linos and V. Courtois, *A Tool for Understanding Object-Oriented Program Dependencies*, Proceedings of the 3rd IEEE Workshop on Program Comprehension, pp20-27, November 14-15, 1994.

[7] M. J. Baker and S. G. Eick, *Space Filling Software Visualization*, Journal of Visual Languages and Computing, Vol. 6, pp 119-133, 1995.

[8] S. G. Eick, *Engineering Perceptually Effective Visualizations for Abstract Data*, In Scientific Visualization Overviews, Methodologies and Techniques, IEEE Computer Science Press, pp191-210, February 1997.

[9] D. F. Jerding and J. T. Stasko, *Using Information Murals in Visualization Applications*, Proceedings of UIST '95 (ACM), November 14-17, 1995.

[10] W. C. Hill and J. D. Hollan, *History-Enriched Source Code*, Computer Graphics and Interactive Media Research Group, Bell Communications Research, Submitted to ACM UIST '93, 1993.

[11] P. Young and M. Munro, *Visualising Software in Virtual Reality*, Proceedings of the IEEE 6th International Workshop on Program Comprehension, pp19-26, June 24-26, 1998.

[12] L. Feijs and R. de Jong, *3D Visualization of Software Architectures*, Communications of the ACM, Vol. 41, No. 12, pp72-78, December 1998.

[13] C. Knight, *Virtual Software in Reality*, PhD Thesis, University of Durham, June 2000.

[14] Java web site http://java.sun.com/

[15] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, *Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization*, Proceedings of the 5th IEEE International Workshop on Program Comprehension, pp17-28, May 28-30, 1997.

[16] A. Von Mayrhauser, A. M. Vans, and A. E. Howe, *Program Understanding Behaviour during Enhancement of Large-scale Software*, Journal of Software Maintenance: Research and Practice, Vol. 9, pp299-327, 1997.