PROVIDING A USER CUSTOMISABLE TOOL FOR SOFTWARE VISUALISATION AT RUNTIME

Michael P. Smith and Malcolm Munro
Visualisation Research Group
Department of Computer Science, University of Durham
Durham, DH1 3LE, UK.
{m.p.smith, malcolm.munro}@durham.ac.uk

ABSTRACT

This paper describes research on developing a customisable tool for visualising object-oriented software at runtime. This aims to highlight both the static and dynamic structure of the software and aid software engineers in tasks requiring program comprehension of the code. The paper specifically looks at some of the customisation support provided by the tool and how a simple representation can support a number of varied tasks.

KEY WORDS

Software visualisation, runtime, dynamic, customisation.

1. Introduction

The task of understanding software is fundamental to the majority of, if not all, software engineering tasks. Development, testing, debugging, maintenance and performance tuning tasks all require some understanding of the software at the source code level. The code provides the static structure of the software and this is essential for understanding the software, however, it can be difficult to get a true understanding from this static description alone. This is especially true for object-oriented software as the paradigm introduces new language ideas which affect its analysis and comprehension. Object-oriented software offers many advantages, however Jerding and Stasko suggest it is "a double-edged sword" [1]. This is due to the discrepancies between the static class descriptions and runtime behaviour as networks of communicating objects [2] [3]. For example, De Pauw et al. state that "There is a dichotomy between the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects) of object-oriented programs. The programmer must understand and map between these structures, a significant burden even after the programmer is familiar with them." [3]. It is due to this that De Pauw et al. state that "Insight into dynamic aspects is critical for understanding, tuning and debugging object-oriented software" [3]. This is the motivation for this research, which aims to improve program comprehension of object-oriented systems by analysing and visualising both their static and dynamic structure through the use of a number of visualisations that provide the user with controls to customise the representation to the user's specific task.

There is a vast amount of information that can be extracted and analysed from a system's static description. However, analysis of its runtime behaviour introduces new information and a huge increase in the available information. Even the analysis of a simple program may lead to a huge amount of data, with details such as method calls and object creation and destruction creating a large and complex information space. It is for this reason that a visualisation approach is used in an attempt to present this large information space in a coherent way and allow the user to drive the analysis and spot patterns and areas of interest. Knight defines this area of software visualisation as "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." [4].

This paper describes the DJVis tool and some of its visualisation and customisation facilities. All of the visualisations and customisation actions are available in real-time as the program under study is executing.

2. The DJVis Tool

DJVis is a prototype visualisation tool designed to show details of Java software as it executes [5][6]. It connects to a running program through a debugging interface (namely the Java Platform Debugger Architecture [7]) in order to extract program events and control the program's execution. The tool is comprised of a number of views, each of which shows some aspect of the program under study. Currently, the main views are: the Runtime View (shows threading and call stack details using a 3D representation [5][6]); the Query View (supports the Runtime View by allowing user controllable grouping and exploration of information [5][6]); the Class View (provides class level details of the software [5][6], as described in the following section); the Method Pixel View (provides details of method calling relationships [5]); and the Variable Watch View (provides a history of read and write accesses to a variable [5]). This paper will specifically focus on the customisation features of the Class View.

2.1 The Class View

The Class View is an essential component of DJVis and is designed to show the software under study in terms of its classes and their structure and relationships. The view uses an augmented graph representation, with the nodes representing types in the software and the edges representing relationships between the types. The nodes are circular and are augmented with additional details about the type. Coming out from the node by default are 'method lines'. Each line represents a method defined by the class and the length and colour of the line represents metrics for that method. Table 1 shows the representations used in the Class View.

Table 1 Class View Representations

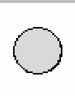
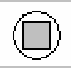
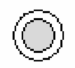
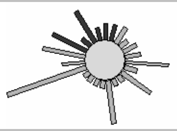
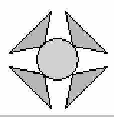
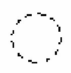
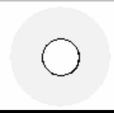
Representation	Meaning
	Class (shading represents metric (number of instances created by default))
	Interface
	Inner Class (inner shading represents metric)
	A Class and its methods (length and shading represent user selectable metrics)
	A class and its fields (shading represents user selectable metrics)
	Type yet to be loaded
	Package type belongs to (Colour coded by package)

Figure 1 shows the representations of four example types within the Class View. The top two types are showing the method length and access rights using the method line length and colour respectively. The XMLReader is an interface which defines a number of methods while the MessageCatalog is a class with a number of long methods and a large number of instances (shown by the dark shading of the class node). The bottom two classes are showing their fields. Both classes have a number of fields and the shading for the DisplayFrame classes represents the access rights of the field while the shading of the GraphCanvas represents the type of the field (primitive type, system class, or user class).

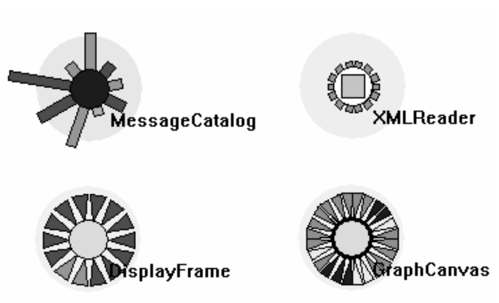


Figure 1 Representing types within the Class View

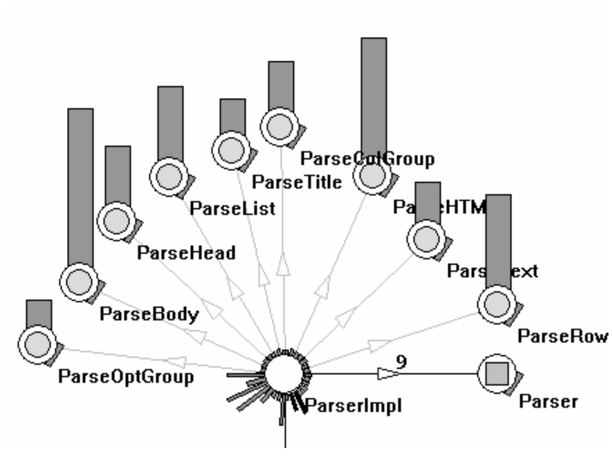
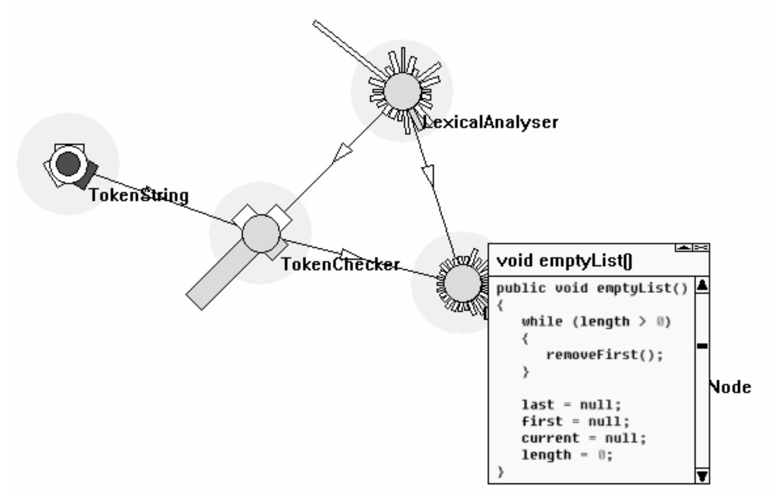


Figure 2 Example of the Class View representation

Figure 2 shows an example of these representations in use. In this example, the method lines are representing the length of the method, while the shading is the access rights of the method (public, protected and private). The edges in the graph represent references between classes with the black edges being static references while the grey edges (actually red in the visualisation) show references through a base type. The shading of the nodes represents the number of created instances of that type. So from this example it can be seen that the ParserImpl class has no instances (as the node is white, further inspection of the code shows that all its methods and fields are static). This class has nine references to the Parser interface (far right), and it has nine dynamic references to the Parse* classes through these variables. The edges can be changed to show the implements relationship, to confirm that all these classes do implement the Parser interface. This representation also allows an overview of the classes to be seen. Each of the Parse* classes has one very short method (in this case the constructor) and one longer method (a Parse method from the Parser interface). The ParserImpl class has a large number of methods, however most of these are very short and inspection using the popup source browser shows that these methods are simply an interface to the Parse* classes.



void emptyList()

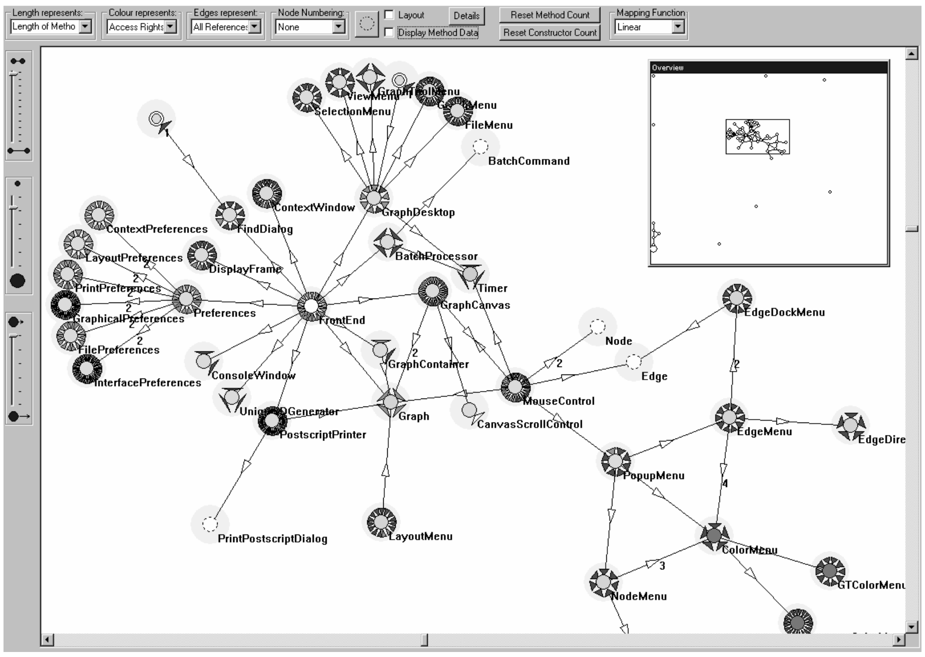
```
public void emptyList()
{
   while (length > 0)
   {
      removeFirst();
   }

   last = null;
   first = null;
   current = null;
   length = 0;
}
```

Figure 3 Popup source browser

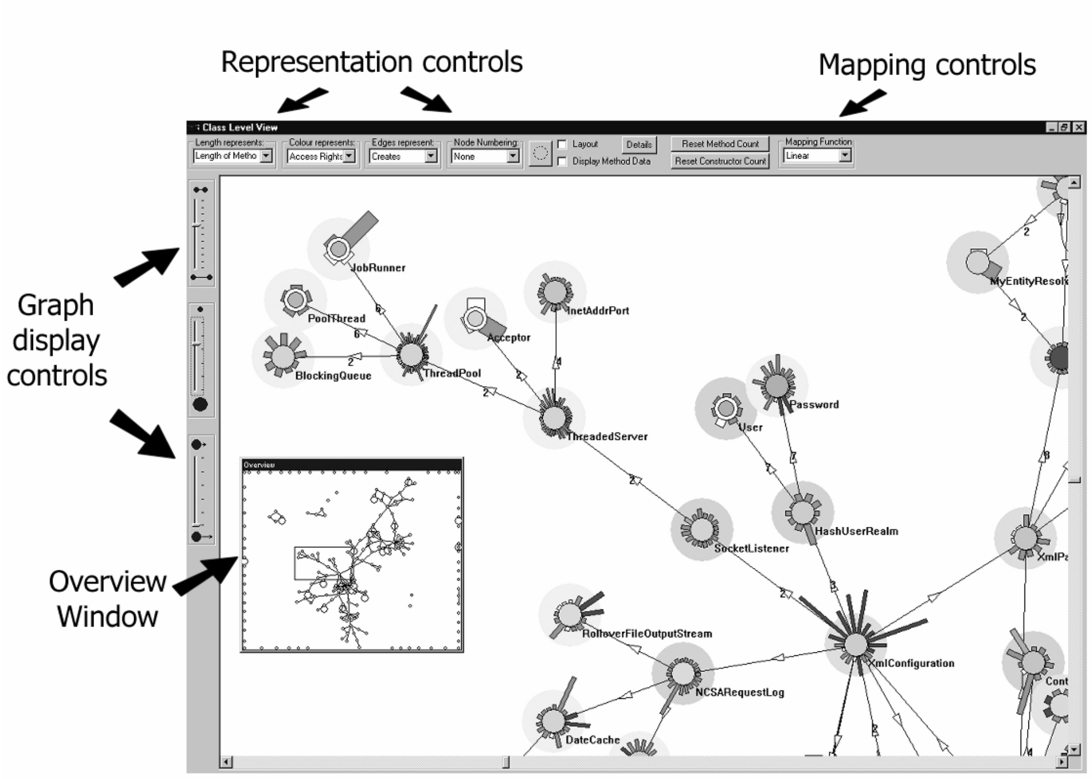
Figure 3 shows an example of the popup source browser. Here a cluster of classes is being inspected. The user is viewing the source code for the emptyList() method which belongs to the class to the left of the browser window. The user can access the popup browser by selecting a method of interest or by hovering the mouse over a method. The popup browser allows the user to quickly inspect the code and test hypotheses that the visualisation may have generated. This also provides a direct link between the representation and the underlying code. For example, in Figure 3 the length of the method lines are representing the length of the methods and the shading is representing the number of calls (the darker the shading the greater the number of calls). It can be seen that very few of the methods have been called for these classes and only a small number of objects of each type have been created (shown by the light node shading). The only exception to this is the TokenString inner class which has had a large number of instances created and one of its methods called frequently. Here the popup browser could be quickly used to show that this is its constructor and to inspect the body of the method if desired.

Figure 4 shows an example of displaying field information. Here the shading represents access rights allowing the user to gain an overview of variable encapsulation and identify any areas with poor encapsulation (i.e. many public fields).



Length represents: Length of Metho
Colour represents: Access Rights
Edges represent: All References
Node Numbering: None
☐ Layout
☐ Display Method Data
Mapping Function: Linear

Figure 4 Overview of variable encapsulation


Class Level View

Length represents: Length of Metho

Colour represents: Access Rights

Edges represent: Creates

Node Numbering: None

Mapping Function: Linear

Figure 5 DJVis using the Class View to visualising an executing web server.

2.2 *Customising the Class View*

This relatively simple representation can display a large amount of information through its support for user customisation. Instead of trying to display all the information at once, the visualisation allows its basic elements to be used to represent a variety of information about the program under study. This is achieved by using drop-down lists at the top of the Class View's main window. These control what the individual elements (method line length and shading, node shading and labelling, and edge types) represent and provide an easy way for the user to read off the current settings, thus preventing user disorientation. The view also offers options for controlling the graph layout and the scaling of the nodes and edges. These settings are controlled by sliders down the left side of the view.

Figure 5 provides an example of the Class View in action and shows the layout of the user interface. Here a web server is being visualised. The edges of the graph are representing the creates relationship and the method lines are showing the length and access rights of the method using the method line length and colour respectively. The dispersion pattern in the graph in the overview window shows the clustered creation patterns with a few key classes at the centre of the clusters and then the outward spread of class creation. The view is zoomed in to see a specific section, with one of the central classes shown (XMLConfiguration (bottom right)) and chains of creation coming out from it (for example the SocketListener chain).

The displayed information can be adapted to specific tasks through the use of custom mapping modes. These map the metrics, such as the number of method calls or method length, before they are applied to the representation. A number of mappings are provided such as linear, logarithmic, scaled and fixed length, however the flexibility comes from allowing the user to define their own mappings to suit a particular task. The mapping functions are defined by graphical manipulation in a manner similar to the 'curves' options in image editing applications. The mappings can then be saved and selected using the drop down list, as with the pre-set mappings.

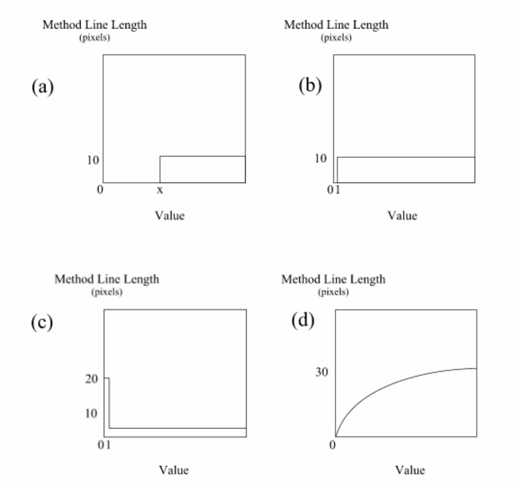
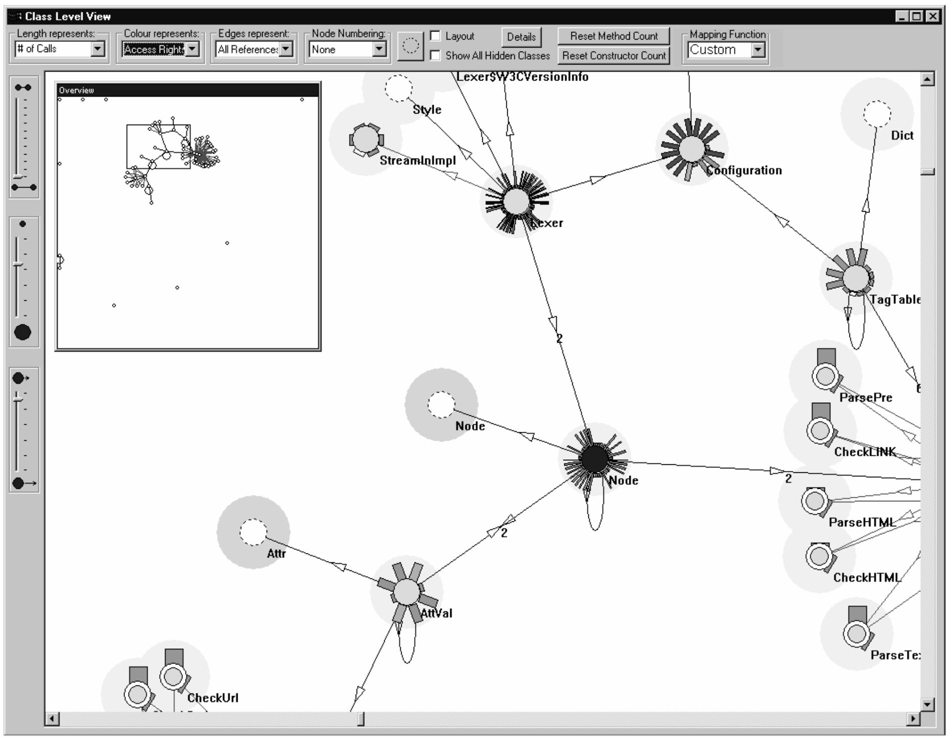


Figure 6 Example of custom mappings for method line length

Figure 6 demonstrates four possible mapping modes that can be defined for the method line lengths. Graph (a) would show only items of values x and over and these would be represented with a constant length of ten pixels. This could be used for example to show methods over a certain size, or to highlight frequently called methods. Graph (b) maps all values to the same value, except zero. This setting could be used to show only those methods that had been called (irrespective of the actual number of calls) or to show nonempty methods (exclude those that are defined but have zero length). Graph (c) shows an alternative case where all values above zero are mapped to a small fixed length while values of zero are mapped to a longer length. This mapping will allow all values to be seen, but will make values of zero prominent. This mapping could be useful for example in a testing setting. Running the program under its normal execution, or over a specific test case will use certain methods of the software. After the execution, this mapping can be used to easily identify classes and methods that have not been covered in the execution. Figure 7 shows an example of this mapping in use where it is applied to the number of calls before they are represented using the method line length. In this example it can easily be seen that a number of the classes have yet to have a number of their methods called. For example, the Configuration class (top right) has only had two of its sixteen methods called. The mapping could also be used to inspect if a particular method was called such as an initialisation method before a specific section of execution. Finally, graph (d) in Figure 6 shows a mapping that would restrict the method lines to a maximum length of thirty whilst making smaller changes more prominent. Mappings such as this and the logarithmic pre-set allows the visualisation to present a diverse range of values whilst preventing large values from obstructing and dominating the visualisation. This is particularly important for showing information such as the number of method calls.

This approach of definable mappings allows for user customisation and it can be applied to any metric that maps to a numeric value. So for example, if support for obtaining method complexity was added to the prototype


Class Level View

Length represents: # of Calls
Colour represents: Access Rights
Edges represent: All References
Node Numbering: None
☐ Layout
☐ Show All Hidden Classes
Mapping Function: Custom

Figure 7 Viewing uncalled methods for a coverage summary

tool then this could be represented using the method line length. Custom mappings could then be used to effectively hide simple methods, while highlighting methods above some threshold of complexity. This would be useful for preventative maintenance tasks where there is a desire to find and improve complex methods.

The mapping functions can also be applied to the shading of items to defined brightness levels dependant on values, for example for the node or method line shading.

Some of the data visualised in the Class View has a temporal nature, for example, method calls are ordered. Such temporal relationships are not explicitly shown in the Class View itself, however the view can give an indication of this data through the use of transparency of items. This is primarily used to represent method calls in order to highlight which classes and methods have recently been involved in the execution. Here, methods that have been called recently are fully opaque while those that have not been called for a long time period appear increasingly transparent. This transition can be controlled through a number of pre-set mappings or through defining a custom mapping function. This is designed to allow the user to spot execution patterns in the view.

3. Related Work

A number of existing approaches have investigated runtime analysis and visualisation of object-oriented software. However, due to the large information space and the varied task set which this information can support, the existing work has only scratched the surface of the potential for program comprehension tasks. Work in this area includes Jinsight [8][3] and Program Explorer [9], which is a program visualiser for C++ developed by IBM research. There has been other research looking at a variety of topics, including the extraction of UML sequence diagrams [10], performance tuning tools [11] and tools to aid teaching and debugging, such as VisiVue^{TM} [12]. The DJVis approach offers a variety of different visualisations and is notable for its focus on the use of user customisation to allow for specialisation.

4. Conclusion

This paper has described the customisation features of the Class View and shown how it allows the visualisation to be adapted to specific tasks. This customisation is at two levels:

1. The user controls which metrics are applied to which representations within the visualisation.
2. The user controls how the value of metrics are mapped before they are applied to the representation.

These two customisation features provide a large amount of flexibility and allow the visualisation to be tailored to highlight specific patterns of interest to the user.

5. References

[1] D. F. Jerding & J. T. Stasko, Using Visualization to Foster Object-Oriented Program Understanding, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical report GIT-GVU-94-33, 1994.

[2] E. Gamma, R. Helm, R. Johnson & J. Vlissides, *Design patterns: elements of reusable object-oriented software* (Addison-Wesley, 1994).

[3] W. De Pauw, D. Kimelman & J. Vlissides, Visualizing Object-Oriented Software Execution, In *Software Visualization* (J. T. Stasko, J. B. Domingue, M. H. Brown & B. A. Price (eds.), MIT Press, 1997).

[4] C. Knight & M. Munro, Visualising Software – A Key Research Area, *Proceedings of the IEEE International Conference on Software Maintenance*, Oxford, England, 1999.

[5] M. Smith, Runtime Visualisation of Object-Oriented Software, PhD Thesis, Department of Computer Science, University of Durham, 2003.

[6] M. Smith & M. Munro, Runtime Visualisation of Object Oriented Software, *Proceedings of the IEEE 1st International Workshop on Visualizing Software for Understanding and Analysis*, Paris, 2002, 81-89.

[7] Java^{TM} Platform Debugger Architecture
http://java.sun.com/products/jpda/

[8] Jinsight
http://www-106.ibm.com/developerworks/library/jinsight/

[9] D. B. Lange and Y. Nakamura, Program Explorer: A Program Visualizer for C++, *Proceedings of USENIX Conference on Object-Oriented Technologies* (COOTS*)*, Monterey, California, 1995, 39-54.

[10] K. Mehner and B. Weymann, Visualization and Debugging of Concurrent Java Programs with UML, *Proceedings of the. Workshop on Software Visualization, International Conference on Software Engineering*, Toronto, Canada, 2001.

[11] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson and J. Isaak, Visualizing Dynamic Software System Information through High-level Models, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications,* (OOPSLA '98), 1998.

[12] VisiVue^{TM}
http://www.visicomp.com/product/visivue.html