The end of the line for Software Visualisation?

Stuart M. Charters, Nigel Thomas and Malcolm Munro
Visualisation Research Group
Department of Computer Science
University of Durham,
South Road,
Durham,
DH1 3LE, UK
S.M.Charters@durham.ac.uk Nigel.Thomas@durham.ac.uk Malcolm.Munro@durham.ac.uk

Abstract

This position paper addresses the issue of how software visualisation should develop in the future. A number of useful visualisations have been developed by the software visualisation community but these have usually been through standalone tools. Is it now time to consider if and how these visualisation tools can be integrated into development environments and be used as roundtrip visualisation tools. If this is not addressed then software visualisation research may have come to a useful end.

1 Introduction

Software visualisation is a relatively young research area where great progress has been made in developing ideas, representations and tools to aid program comprehension during the maintenance and evolution of software. This development is paralleled in the software development community where visual representations of systems have been used to help in for example requirements capture and design [11] [7]. The representations and tools to aid program comprehension take a variety forms from animations of algorithms and data structures, through dynamic run-time information, to tools which present static view of software structures linked to source code views [8] [9] [10] [3] [2] [4]. Despite these advances it is time to question if the time has come when no new advances are being made and all advances are just variants on the old theme.

Software development is an evolutionary process, often one of iterative refinement. Segments of code are written, tested, refined and built upon. Software visualisation needs to support this iterative process, if tools are stand alone the effort to evaluate changes to code using those tools is considerable.

In an ideal world the maintenance and evolution of a system is carried out on the appropriate level of structure within the lifecycle model because of the in built traceability between the documents that result from each phase. For example, corrective maintenance is concerned with the code, and changes are made at this level, whereas perfective maintenance is concerned with changing requirements and hence the changes should be made to the requirements and then be reflected through to changes in the design documents and the code. In practise however, maintenance and evolution is carried out at the code level because the traceability has been destroyed through excessive maintenance or it never existed in the first place. In this situation changes are rarely reflected in the other lifecycle documents that define the system. Thus the need for program comprehension supplemented by visualisation.

It is recognised that program comprehension occurs in different ways, top-down, bottom-up or a combination of the two [12]. This program comprehension can be systematic or as-needed, the integration of visualisation allows developers and maintainers to use whichever strategy is required or suits them best to achieve the understanding they require to make changes. Another view of program comprehension is the feedback loop strategy, where the program is compared against the mental model of the problem solution held by the developer. The use of visualisation throughout the implementation phase would complement this feedback loop strategy as the developer saw the program growing visually allowing them to continually compare this against their mental model. Work has already been done to integrate different views to allow the use of different comprehension strategies within a number of tools [9].

2 A Simple Analogy

This section draws a simple parallel between the development of HTML pages and program code. It is not intended to be a comprehensive comparison but to act as a simple analogy to illustrate a possible way forward for software visualisation.

When developing an HTML page one approach is to use a simple text editor to write raw HTML and then to view that HTML in a browser. In the maintenance of that HTML page the maintainer will iterate between the editor and the browser, always changing the HTML in the editor. The use of the browser can been seen as using a visualisation of the HTML code to check that it is correct and to get some understanding of how the HTML works. This type of use is termed a one-way trip, in that the code (HTML in this instance) is edited and the visualisation is used to give some understanding. This can be seen as a simple analogy with one way that program code is developed and where visualisation is used to help understand some aspect of that code. The programmer will use simple visualisation tools (such as call graphs and control flow diagrams) to supplement their understanding of the program source code.

Another way to develop HTML pages is to use a development environment such as Dreamweaver. Here the main interface is a visual one that allows WYSIWYG layout and editing of the underlying HTML without having to resort to understanding the HTML. Changes in the visual interface generate or update the underlying HTML. In addition these development environments allow the direct editing of the underlying HTML and changes made via the textual representation are reflected in the visual interface. This type of use is termed roundtrip editing.

Of course it is recognised that Dreamweaver is a tool with complete integration that operates on a somewhat restricted language (HTML) that is inherently visual. The Dreamweaver type of tool [5] [1] [6] works with HTML because there is a one to one mapping between the HTML tags and the visual representation. With software visualisation tools this one to one mapping is less obvious and harder to achieve due the complexity of programming languages and the nature of the visualisations.

Visualisation of software systems show the relationships between the components of the system at different levels of granularity and at different levels of abstraction. They come in many forms and range for example, from high-level architectural representations, through design notations (UML) to structural representations such as call graphs and control flow diagrams. To these may be attached attributes that show, for example, the relationships between names (variable, class etc.) used in systems. These visualisations have been shown for example, using conventional node and arcs, tables, and grids, and have utilised real life and abstract metaphors in a two-dimensional or a virtual reality world.

3 Roundtrip visualisations

Roundtrip visualisation is used to describe visualisation systems that are linked with the data from which they are generated in such a manner that changes to the underlying data updates the visualisation and changes made through the visualisation itself are reflected in the underlying data. An example of roundtrip visualisation for software is the construction of a visualisation that represents the class structure of a Java project and where if the structure of the classes is modified then the visualisation is updated and similarly if the visualisation is used as a mechanism to restructure classes then the code reflects that restructuring.

Current visualisation tools tend to be one-way trip. One model is where the source code is edited and this is reflected in the visualisation but not the other way round. An example of this is a call graph visualisation linked to a source code editor that changes as the calling structure of the code is modified. A further one-way trip mode is where the visualisation is edited and this is reflected in the code but not the other way round. An example of this is the JBuilder GUI designer where the visualisation consists of a canvas and a palette of GUI components. The canvas can be 'edited' in order to change the GUI and these changes are reflected in the java source code. However if the generated GUI java code is edited directly then the changes are not necessarily reflected on the canvas.

The limiting factor is that the visualisations developed so far do not have the required properties for roundtrip visualisations. The current visualisations of software are at the wrong level of abstraction or of the wrong granularity and thus are one-way trip visualisations.

4 Conclusion

Current progress in software seems to confined to:

- improving abstractions to reduce information overload;
- developing new representations using abstract or real world metaphors; and
- improving layout of existing representations;

and are instantiated in one-way trip standalone tools. These are all laudable research aims and can sustain software visualisation research for a while longer.

The way forward for software visualisation is to address the issues of roundtrip visualisation. To support roundtrip visualisation an alternative approach is required, partial integration with the development environment is needed to allow for access to the source data by the visualisation and for changes made in the visualisation to be reflected in the development environment. However this integration must be sufficiently flexible, for example by the use of a standard integration method for visualisations, that different types of visualisations can easily be integrated and that visualisations can be integrated with different development environments. The roundtrip nature of the integration would need to ensure that changes made using the visualisation are reflected in the source and that changes to the source were reflected in the visualisation.

If this issue is not addressed then it really is the end of software visualisation. We must develop new visualisations that can easily integrate as roundtrip visualisations.

References

[1] Adobe. Pagemill. *http://www.adobe.com/*, 2003.

[2] J. Cain and R. McCrindle. Software visualisation using c++ lenses. *Proccedings of 7th International Workshop on Program Comprehension*, May 1999.

[3] C. Knight and M. Munro. Comprehension with(in) virtual environment visualisations. *Proccedings of 7th International Workshop on Program Comprehension*, May 1999.

[4] C. Knight, M.-A. Storey, and M. Munro. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[5] Macromedia. Dreamweaver. *http://www.dreamweaver.com/*, July 2003.

[6] Netscape. Netscape composer. *http://www.netscape.com/*, 2003.

[7] P. W. Parry, M. B. Ozcan, and J. I. Siddiqi. The application of visualization to requirements engineering. *Technical report, Computing Research Centre, Shefield Hallam University, England*, 1998.

[8] M. P. Smith and M. Munro. Runtime visualisation of object orientated software. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[9] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. *In Proceedings of the IEEE Symposium on Information Visualization*, 1997.

[10] C. M. B. Taylor and M. Munro. Revision towers. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[11] A. Teyseyre, R. Orosco, and M. Campo. Requirements visualization. *Workshop de Investigadores en Ciencias de la Computacin(WICC'99)*, 1999.

[12] A. Von-Mayrhauser and A. M. Vanns. Program comprehension during software maintence and evolution. *IEEE Computer*, August 1995.