Managing Views in a Program Understanding Tool

Kenny Wong *

Abstract

Program understanding tools typically offer built-in visual representations of the subject software, such as call graphs and class hierarchies, and textual representations, such as cross-reference listings and exact-interface reports. It is useful to bundle a number of these visual and textual frames, with some annotation, into a *view* for redocumentation purposes. For large, legacy software systems, however, the abundance of created views can be a major problem. This paper investigates a number of methods for improving the organization of these views for improved usability and scalability.

1 Introduction

Many software systems have internal documentation that is often out-of-date and thus unreliable. Even when the documentation exists, it may be dispersed in several places and may not be well structured. Yet accurate, complete, well-organized, and maintainable documentation is critical for software maintenance. As such, a programmer often falls back on the source code to understand how the software works, spending much time and perhaps duplicating the efforts of others. This problem becomes more critical when maintaining large, legacy software systems. Thus, there is a need for "redocumentation" [5].

Reverse engineering is one way of redocumenting an existing software system. This process identifies software building blocks, extracts structural dependencies, produces higher-level abstractions, and presents pertinent summaries for maintenance and reengineering (program understanding) purposes. These summaries may be presented in visual or textual *frames* that are derived from information in a centralized repository. A visual frame might contain, for example, a high-level overview of the system, a function call graph, an inheritance hierarchy, or a control/data flow chart. A textual frame might contain, for example, software-metric results, a cross-reference listing, or an exact-interface report between two modules. A frame displayed in a window need not be static, but can be fully interactive. For example, one can adjust a module interconnection graph in a visual frame by grouping, arranging, filtering, scaling, clustering, and editing operations [13]. One can select a function listed in a text frame and link to additional documentation that describes the function in more detail [12].

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

Reprinted from *Proceedings of CASCON '93*, (Toronto, Ontario; October 25-28, 1993), pages 244–249.

This auxiliary documentation might be external to the repository. Moreover, the frames might be derived from many different tools.

Pictorial, animation, audio, and video frames can serve as annotation. Frames can be bundled and saved into a *view* for redocumentation purposes [13]. The usability of views requires that the constituent frames have a well-defined "procedural" representation and be dynamically updated. The procedural representation provides a parameterized script or specification for regenerating or retrieving the frame. The script also allows the more flexible possibility of adjusting, filtering, or even modifying the contents of a frame prior to presentation. Dynamically-updated frames refresh themselves to accurately portray the appropriate subset of the repository (or auxiliary documentation) at all times. Thus, unlike traditional software documentation, a view remains consistent and up-to-date.

These views can be organized into *global* views for the workgroup and *local* views for the individual user. Moreover, the relevant views can be assembled and targeted for different classes of users (designers, programmers, and managers), different levels of experience (beginners versus experts), and different kinds of purposes (guided tours, overviews, and details). This improves program understanding and eases decision-making by programmers and especially managers [13]. Thus, views are a high-level abstraction for incorporating more application and domain-specific types of knowledge.

2 Organization

Further structuring mechanisms are necessary to manage the large number of created views that is needed to document a large software system.

2.1 Subviews

Views can be enhanced to contain a pointer to other views, forming a *subview* relationship between views and building a multiple inheritance/specialization hierarchy of views. This idea allows views to be tailored or specialized for different interpretations more easily and to share effort in their construction in a way similar to object-oriented programming. A manager may want the same high-level overview as a programmer, but with additional annotation frames to present, for example, personnel assignments, funding levels, and scheduling information. The regeneration of a view becomes recursive; changing a view updates the views to which it is a subview.

Views can already "contain" other views by merely including the constituent frames of the other views. However, this aggregation approach, while simple, seems less suitable and less flexible than the inheritance approach, because of the lack of an explicit structure.

2.2 Sequenceable views

Analogous to hypertext/hypermedia, links can be used to associate arbitrary views (and frames), forming a web of views. However, for a large enough number of views, there are significant problems with user disorientation during navigation and cognitive overhead for managing the links [4].

One solution to these problems is to use the concept of a *path*, an ordered traversal of some links in the web [15]. The notion of a *sequenceable view* unifies subviews and sequential paths. The top-level views and frames of a sequenceable view are linearly ordered and played back one at a time. When encountering a sequenceable view during the playback, the user can be given the choice to skip past it or to "dive" into it. This capability provides a simple roaming and zooming mechanism as well as a branching technique. Note that there could be a mix of unordered and sequenceable views in the hierarchy.

2.3 Template views

To maintain consistency and completeness between different sets of views, possibly for differing software versions and variants, there is a need for standardized sets of views. The notion of *templates* [6, 10] or frameworks for standardizing software documentation and hypertext structures can be applied to views. Another benefit is that template views promote reusability.

A parameterized script representation for frames is needed. Template views are instantiated into real views by binding the parameters of the constituent frames appropriately.

3 Presentation

There are two issues: presenting the structure of the view documentation and presenting the frames of the views themselves. This section mainly focuses on the first issue.

3.1 Inclusion graphs

To enhance understanding and promote insight, visual representations of software are generally desirable [2]. Thus, a visual method for presenting the structure of a set of views is needed. One promising approach is to use inclusion graphs where views and frames are represented by labelled boxes (rectangles) and the subview relationship is represented by the recursive nesting of boxes (see Figure 1). The chief advantage of this "space-filling" approach to visualizing and browsing hierarchical structures, versus a tree or graph drawing, is the efficient use of display area [9]. This technique has been used for visualizing designs, where smooth navigation among drastically different levels of abstraction is required [7]. With some enhancements for position, size, color, and font cues, the technique has also been used for visualizing very large knowledge structures in a "virtual museum metaphor" [14].

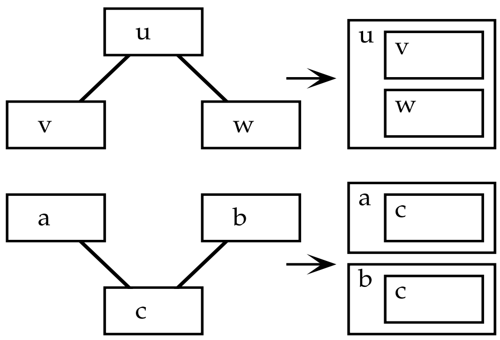

Figure 1: Inclusion Graphs

3.2 Canvases

The inclusion graph, up to some depth, can be rendered in a single window on the screen. The window or *canvas* becomes the working area and focus of related view construction, editing, and navigation activities. Multiple canvases are possible; each could capture a different set of environmental and configuration possibilities of the subject software. To symbolize a frame by a new box (or miniature [11]) in a canvas, one might drag an indicator from the frame and drop it onto an appropriate spot on the canvas, perhaps in the box of an existing view (see Figure 2). Views begin as empty boxes and can be edited by dragging and dropping the boxes of frames and views inside each other. Boxes can be "opened" to generate, playback, and present their contents (frames and/or views). Boxes can be "raised" to allow deeper structures to be rendered and "folded" to mask their structure.

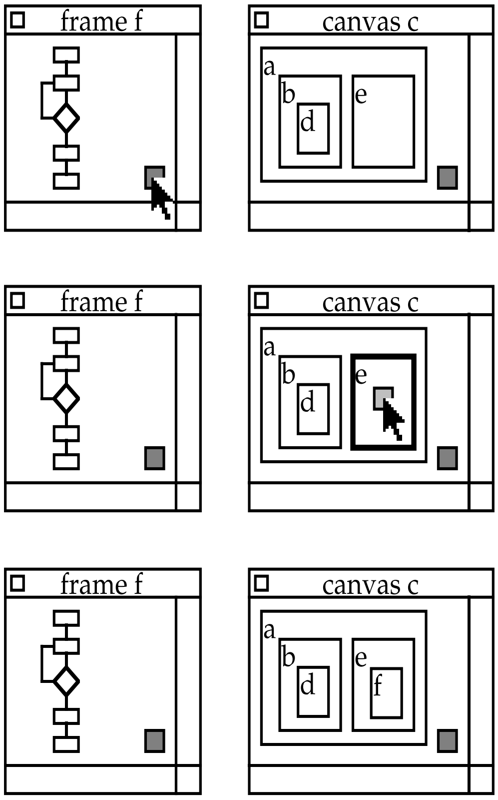

Figure 2: Adding a Frame to a View

3.3 Metaviews

It is instructive to think of a canvas as simply another kind of frame that provides a high-level "map" for a number of views. Bundling one or more of these canvas frames into a view forms a *metaview*; symbolizing one or more of these metaviews on a canvas forms a *metacanvas*. The number of levels of abstraction is unlimited. This allows the organization of views to be scalable, a necessity for understanding large software systems. One could, for example, use a metacanvas to design a map that shows the different sets of view documents for different releases of the software (according to different baselines of the repository). This is useful to separate and manage the program understanding analyses.

3.4 Layout

Since each frame is typically displayed in a window, it becomes tedious and, eventually, unmanageable to lay out the frames of a view by hand. This becomes critical for a large number of changing views. Thus, automatic and adaptive layout is an important requirement that needs further research [3].

4 Summary

This paper investigates and proposes various ways to improve views. For large software systems that require many views to fully document them, some kind of organization scheme is needed. Subviews, sequenceable views, and metaviews help to structure the view documentation with unlimited levels of abstraction. Template views promote reuse and allow for consistent, standardized views. Canvases present a compact, visual representation of the organization of views.

About the author

Kenny Wong is a PhD student in the Department of Computer Science at the University of Victoria. His research interests include program understanding, user interfaces, and software design. He is a member of the ACM, USENIX, and the Planetary Society. His internet address is `kenw@sanjuan.uvic.ca`.

References

[1] Penny Bauersfeld, John Bennett, and Gene Lynch, editors. *Proceedings of the ACM Conference on Human Factors in Computing Systems*, Monterey, California, May 1992. ACM Press.

[2] Heinz-Dieter Böcker, Gerhard Fischer, and Helga Nieper. The enhancement of understanding through visual representations. In Marilyn Mantei and Peter Orbeton, editors, *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 44–50, Boston, Massachusetts, April 1986. ACM Press.

[3] Grace Colby. Maintaining legibility, structure, and style of information layout in dynamic display environments. CHI'92 Posters and Short Talks, May 1992.

[4] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.

[5] Nigel T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *Proceedings of the Conference on Software Maintenance*, pages 54–59. IEEE Computer Society Press, 1988.

[6] Pankaj K. Garg and Walt Scacchi. A hypertext system to manage software life-cycle documents. *IEEE Software*, pages 90–98, May 1990.

[7] Raymonde Guindon. Requirements and design of designvision, an object-oriented graphical interface to an intelligent software design assistant. In Bauersfeld et al. [1], pages 499–506.

[8] *Hypertext '89 Proceedings*, Pittsburgh, Pennsylvania, November 1989. ACM Press.

[9] Brian Johnson. TreeViz: Treemap visualization of hierarchically structured information. In Bauersfeld et al. [1], pages 369–370.

[10] Daniel S. Jordan, Daniel M. Russell, Anne-Marie S. Jensen, and Russell A. Rogers. Facilitating the development of representations in hypertext with IDE. In HYPERTEXT89 [8], pages 93–104.

[11] Jakob Nielsen. Miniatures versus icons as a visual cache for videotex browsing. *Behaviour & Information Technology*, 9(6):441–449, November-December 1990.

[12] Scott R. Tilley and Hausi A. Müller. INFO: A simple document annotation facility. In *SIGDOC '91: Proceedings of the 9th Annual International Conference on Systems Documentation*, pages 30–36, Chicago, Illinois, October 1991. ACM Press.

[13] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting soft-

ware systems with views. In *SIGDOC '92: Proceedings of the 10th Annual International Conference on Systems Documentation*, pages 211–219, Ottawa, Ontario, Canada, October 1992. ACM Press.

[14] Michael Travers. A visual representation for knowledge structures. In HYPERTEXT89 [8], pages 147–158.

[15] Polle T. Zellweger. Scripted documents: A hypermedia path mechanism. In HYPERTEXT89 [8], pages 1–14.