Documenting Software Systems with Views III: Towards a Task-Oriented Classification of Program Visualization Techniques

Scott Tilley
Department of Computer Science
University of California, Riverside
stilley@cs.ucr.edu

Shihong Huang
Department of Computer Science
University of California, Riverside
shihong@cs.ucr.edu

ABSTRACT
Documentation has long played a key role in aiding program understanding. Graphical forms of documentation rely on software visualization techniques to make complicated information easier to understand. However, it is an open question exactly which types of graphical documentation are most suitable for which types of program understanding tasks (and in which specific usage contexts). This paper describes preliminary work towards a task-oriented classification of program visualization techniques. The classification is currently descriptive in nature, and divides the visualization techniques into three classes (static, interactive, and editable) based on the level of end-user interaction with the generated graphical documentation. The primary advantage of a task-oriented classification is that it will ultimately map common activities related to program understanding to specific types of software visualization. A summary of how the descriptive classification was used to structure the selection of software visualization tools to support program understanding in an industrial context is provided.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Maintenance, and Enhancement – Documentation, Reverse Engineering

General Terms
Documentation, Human Factors

Keywords
Documentation**,** program understanding, visualization

SIGDOC'02, October 20-23, 2002, Toronto, Ontario, Canada.

1. INTRODUCTION
Legacy systems represent substantial corporate investments that must be continually enhanced to meet new requirements. Evolving such complex systems requires an understanding of the program's current functionality and structural properties. Documentation has long played a key role in aiding program understanding to support this evolutionary process. Software engineers rely on program documentation as an aid in understanding the functional nature, high-level design, and implementation details of complex applications. The lack of high-quality documentation is particularly acute for large-scale systems, since gaining a sufficient understanding of their singular complexities is an extremely challenging task.

There are two broad categories of documentation that are used as an aid in program understanding [8]. The first category is textual, exemplified by printed manuals and user guides. A more flexible form of textual documentation is electronic, such as HTML or XML files, which permit activities such as automated indexing and the creation of hypertext links between document fragments.

The second category of documentation is graphical, exemplified by charts and graphs. Like textual documentation, a more flexible form of graphical documentation is also electronic, which permits actions such as interactive layout and user-directed editing of the graphs. Graphical documentation relies on a variety of software visualization techniques to make complicated information easier for the developer to understand (e.g., highlighting complex dependencies in a large application by drawing nodes, arcs, and related artifacts on a computer display).

Software visualization tools are often advocated as an effective means of aiding program understanding by creating graphical representations of the subject system "to reduce complexity of the exiting software system under

consideration" [10]. While there is no doubt that such visual images can be artistically pleasing to the eye, there is little scientific evidence that such forms of graphical documentation are superior to textual documentation in effectively aiding program understanding. There are many unresolved research issues related to which types of diagrams are most appropriate for aiding program understanding. More specifically, it is unknown exactly which forms of graphical documentation are most suitable for which types of program understanding tasks, and in which specific usage context.

As part of a larger collaborative project with an industrial partner, we have begun to address this fundamental question. The project's primary goal is to create a methodology for the reuse of legacy software. One of the first steps towards reuse is the redocumentation of the software system to aid in its understanding. To support this goal, the project has a secondary goal of constructing a prototype toolset supporting the methodology. A key requirement of this toolset is the ability to visualize various aspects of the software system under scrutiny in several different ways. This requirement led to an extensive investigation into the capabilities of current software visualization tools, and the creation of a comparison framework for evaluating the tools' capabilities in the context of the industrial partner's numerous constraints.

The ultimate goal is to use this comparison framework in conjunction with a task-oriented classification of program visualization techniques to better guide the tool selection process. The advantage of a task-oriented classification is that it maps common activities related to program understanding, such as "show me which variables this function uses", to specific types of software visualization, such as reference diagrams.

However, creating such a general task-oriented classification framework is a significant undertaking. Therefore, we have begun to address the problem by first creating a descriptive classification of program visualization techniques. The main axes of this classification are determined by the level of interaction between the user and the generated graphical documentation.

It is important to note that this descriptive classification is a means to an end, not an end in itself. Moreover, the classification descriptors are heavily influenced by the somewhat unique characteristics of the industrial partner's application domain: real-time, embedded control systems. Nevertheless, the preliminary version of the classification framework does represent an important step towards the goal of creating a task-oriented classification of program visualization techniques.

The next section discusses some of the fundamental issues related to software visualization. Section 3 presents the descriptive classification of program visualization techniques. Section 4 illustrates the use of this classification system in support of software visualization tool selection in a real-world project. Finally, Section 5 summarizes the paper and outlines possible avenues for future research.

2. SOFTWARE VISUALIZATION

> *"The computer displayed a fractal graph of an amino-acid distribution in the genetic structure. She'd chosen this type of fractal representation because it was a technique that enabled the human eye and mind to quickly grasp the totality of an individual's genetic code. Seeing the visual relationship of the code's different amino acids was infinitely more efficient than spending hours viewing the complete, base pair-by-base pair listing of a human genome.* ***Pictures and patterns, on the other hand, were what the human mind had evolved to understand****."* [13]

There are many definitions of software visualization that depend on the intention of the authors or the purpose of the visualization being used. One definition that focuses on the software aspect is defined in [7], which states that software visualization is a special type of information visualization that uses computer graphics and animation to help illustrate and present computer programs, processes, and algorithms. There are many different applications of software visualization, such as object-oriented systems (class browsers), data structures (compiler data structures in particular), real-time systems (state-transition diagrams, Petri nets), data flow diagrams, subroutine-call graphs, and entity-relationship diagrams (UML and database structures) [9].

Graphical representations of complex data have long been incorporated in analytical software tools in numerous technical areas [4]. Consequently, the basic data model underlying most software visualization systems is a directed graph. Nodes in the graph correspond to software artifacts, such as functions, data types, and subsystems. Directed edges correspond to function calls, data dependencies, and containment relationships. Software visualization tools are used to explore the internal

hierarchical structure of the underlying software system, as represented by the graph.

For those who tend to think visually, graphical displays of the software system may greatly increase their understanding of the program. There are many factors that affect the suitability and efficacy of the generated graphics in terms of aiding program understanding, such as the layout algorithm used, edge routing, focal points, node/edge colors and shapes.

2.1 Related Work

There is a significant amount of ongoing research into software visualization techniques. There is considerable overlap between this area and related work in areas such as graph theory and information visualization. Unfortunately, there has been a lamentable lack of evaluation of such software exploration and program visualization tools [14].

A few researchers have conducted experiments to observe how software engineers explore program information. For example, Storey *et al* explored the integration of various visualization techniques in software exploration tools [15]. Koschke conducted an online survey regarding the perceived needs of software engineers with respect to visualization tools [11].

Buchsbaum *et al* have performed very interesting work with an industrial collaborator regarding the effective use of visualization techniques as an aid in understanding software infrastructures [5]. The results from this work help form the general requirements of software visualization tools for the purpose of program understanding.

2.2 Documenting Software Systems with Views

Some of our own work in the area of program understanding has been directly related to the use of views as forms of graphical documentation. "Views" in this context are visualizations of software systems that support multiple perspectives. The views are produced through reverse engineering, which is the process of analyzing the source code of an existing software system and recreating design-level information in an automated manner [6]. Reverse engineering can aid the software engineer in understanding the subject system by providing program documentation where none existed before.

A reverse engineering methodology for documenting a software system with views produced by building subsystem structures out of software building blocks is described in [18]. The graphical views are produced through a semi-automatic reverse engineering process. The views produced primarily addressed the needs of the software engineer and technical manager as document users.

A more recent project on graphical documentation is presented in [8]. An integrated approach to documenting software systems based on standardized representations such as XML and commonly-available visualization tools is described. The specific usage scenario is the documentation process for automotive systems. However, the approach is sufficiently general that it could be applied in other application domains.

2.3 Fundamental Issues

In [17], issues related to selecting software visualization tools for program understanding in an industrial context are discussed, focusing on adoption concerns of research prototypes by commercial enterprises. However, irrespective of how the graphical views are produced, there is an underlying issue that remains unaddressed: Which types of visualization are most supportive of program understanding tasks? This is a fundamental question that should be addressed by the technical communication community. Otherwise, no matter how impressive the visualization technique, if it does not serve a useful purpose in this context it will remain a technical novelty rather than an indispensable aid.

This paper continues the work on graphical documentation by working towards a task-oriented classification of program visualization techniques. By classifying the visualization techniques according to the types of program understanding tasks they support, we can move closer to answering the underlying question regarding the suitability and efficacy of graphical forms of documentation.

As an intermediate step towards this broader goal, we have created a descriptive classification of program visualization techniques. This descriptive classification is described in the next section.

3. A DESCRIPTIVE CLASSIFICATION

There are many ways to classify program visualization techniques. For example, one way is to classify them based on their supporting technology. Another way is to classify them based on the types of visualization techniques and graph layouts they provide.

However, for the purposes of supporting program understanding, we have chosen to partition program visualizations based on the level of interaction between the end-user and the generated graphical documentation. The three classes are static, interactive, and editable. This is not a strictly disjoint classification (i.e., one tool could be in more than one category), but most tools fall more heavily into one category than another with little effort.

There is another aspect of end-user interaction with the program visualizations that should be mentioned. This is the guidance a user might provide to the visualization tool during the generation of the graphical documentation. For example, indicating which portions of the source code to analyze to produce a data flow graph, irrespective of how the user will ultimately interact with the graph. This type of user interaction is an important element in the whole program understanding scenario, but it is not a separate part of the descriptive classification described here.

3.1 Static

Static graphs are the oldest form of graphical documentation. They are relatively easy to produce and to integrate into current software engineering processes. Static visualizations are hardcopy-like images that the user can view or print. The produced graph can be in common formats such as GIF or PDF. Other than possibly selecting the artifacts that appear in the visualization, the user does not interact in any way with the static image; it is included solely as read-only graphical documentation.

Static visualizations are best suited to common program understanding tasks such as function and variable access diagrams, high-level representations of overall software architecture, files and functions dependency, interfaces among components, and so on. Their main advantage is that the generated graphs can be inserted into a printed textual manual that documents the subject system. Different views of the system, at different levels of abstraction, can be produced and captured as static graphs. These graphs can be useful additions to the existing textual documents.

The main disadvantage of static visualizations is that they provide a fixed view of the selected artifacts. This view reflects the author's perspective of the documentation needed to understand a particular software application, which need not necessarily be the view shared by the documentation's users. Moreover, that static nature of the documentation implies that users are passive recipients of the information; they have little exploratory capability – an activity that is often very important for program understanding.

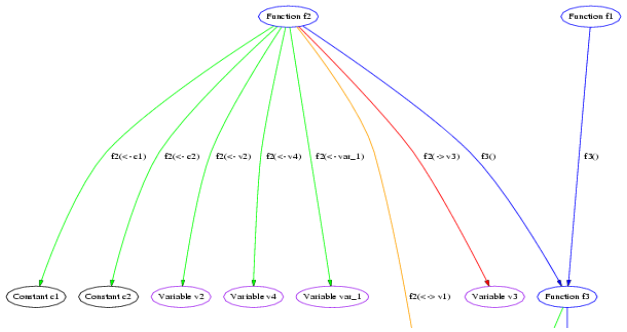

Figure 1: Static Graphical Documentation

An example of static graphical documentation is shown in Figure 1. In a relatively small amount of space, this (simplified) graph captures several different types of information. One colored arc represents different forms of variable access (read, read-write, write) by a function. Another colored arc represents control flow between functions. Both types of information are commonly sought by software engineers when performing maintenance tasks that require an understanding of low-level code details.

3.2 Interactive

The second category of graphical documentation is interactive, which permits the user to navigate the diagram much like a Web page. The user can traverse the graph by selecting an edge and then following the link to the next node. This traversal feature provides a clear path in a complex graph so that the software engineer can chase down the artifacts and relationships of interest.

During the traversal, the fundamental structure of the graph remains unaltered. Some tools allow the user to annotate the graph with their own comments and notes. These are ancillary structures superimposed on top of the graph topology produced by the visualization tool. An example of such post-generated annotation is a Wiki Web [21].

Some visualization tools may also permit the user to change the layout of the graph. For example, changing from a Sugiyama layout to a fisheye view of a subgraph. Alternate graph layouts can be useful in improving comprehension of the program represented by the nodes and arcs – but exactly which types of layout are best, and in which circumstances, is not known.

An important feature of interactive graphs is the ability to model nested graphs using hyperlinks. This can also be realized as a folding/unfolding feature, as a form of graph abstraction. It reduces complexity by collapsing subgraphs of related nodes and edges into single node. This action is particularly useful for visualizing software architecture-level information.

It is commonly believed that interactive graphs provide an improvement over static graphs in terms of aiding program understanding by letting the user explore the system in a manner that best suits their cognitive models and particular maintenance tasks. However, this perceived benefit comes at the cost of increased system support and more complex integration issues. There is a clear need to validate this hypothesis in a more rigorous manner.

3.3 Editable

The third category of graphs produced by software visualization tools is editable. Editable visualizations let the user actually change the generated graph itself, such as adding new nodes or edges. If the editable graph is connected to a backend database, it can be saved back to database for future reference, completing the cycle from maintenance back to new development.

Editable graphs are sometimes referred to as "live documents," since they can react to user input to provide alternate views and/or ancillary information not immediately available in a static graph. Visualization systems that supported live documents were traditionally quite difficult to engineer, due to the extensive programming required to construct the graph editing infrastructure. However, newer developments such as Scalable Vector Graphics (SVG) [20] have made it easier for even modestly-equipped research teams to create live document editors.

Editable graphical documentation carries considerable overhead. It is in effect a form of end-user programming, which always requires more skill and training on the part of its user to fully exploit its capabilities. There is as yet little proof that the stated benefits of editable documentation outweigh its increased adoption cost.

4. AN EXAMPLE

To illustrate the use of the interim classification of program visualization techniques described in Section 3, this section provides a summary of a visualization tool selection case study. The case study was related to the construction of a reverse engineering environment for an industrial partner. Several software visualization tools, both academic and commercial, were analyzed according to several factors. One of the most important considerations was the tools' support of one or more of the three types of visualization techniques: static, interactive, and editable. The relative importance of these techniques varied depending on the nature of the specific program understanding task.

4.1 Program Understanding Issues

There were numerous project-specific requirements for the software visualization tool. These requirements were driven in part by the nature of the project, both in terms of the application domain of embedded real-time control systems, and the software system itself, which imposed special considerations that had to be addressed. These requirements implicitly created considerations related to the nature of the program understanding tasks that the visualization tool had to support.

Many of these considerations were influenced by the overall project goal of supporting code reuse through automatic document creation using reverse engineering technologies. For example, determining and subsequently visualizing code fragments that are dependent on the operating system.

Stringent constraints were placed on the tool selection process due to the unique characteristics of a real-time embedded control system. For example, the selected tool(s) had to have the ability to identify hardware-specific software components, incorporate real-time data gathering into its analysis, and trace timing dependencies from high-level tasks down to low-level bit variables. Most importantly in this context, the result of this analysis must be suitable for visual documentation.

The following is a representative list of some of the project-specific issues that drove the nature of the program understanding tasks to be carried out, and that influenced the software visualization tool selection.

Source Code Analysis

Since the source code of the subject system is written in a highly-optimized manner (to overcome some of the limitations of the implementation language), the engineers' questions are often related to low-level issues. For example, a common task is to understand how bit variables are referenced (read, write, or both). This information should be immediately available to the user, so that race conditions are avoided.

Software/Hardware Interface Analysis

The software under analysis is a real-time control system that runs on an embedded platform. This naturally means the engineers must understand the software/hardware interfaces very well. They need to know which software components are directly reliant on special-purpose hardware. This is because there may be a long-term goal of replacing special-purpose (and more expensive) hardware devices with more generic hardware devices. Switching to commercial-off-the-shelf hardware is cost-advantageous, but only if the software can be reengineered to work properly as well.

Runtime Behavior

One of the most difficult aspects of real-time systems is related to timing considerations. For example, the engineer must understand which parts of the software are calculated in the time-dependent tasks and which parts are calculated in the angle-dependent sequences. A related question is which function(s) run in which task(s)? This knowledge is difficult to glean from manual browsing of source code; it is much better if it can be depicted in a graphical manner by the program visualization tool.

4.2 Tool Selection

The evaluation of the visualization capabilities of the candidate tools was based on an existing framework for assessing the capabilities of reverse engineering tools [16], which was suitably augmented to address the constraints imposed by the project's singular requirements. The commercial software visualization tools evaluated as part of this project were dot from AT&T Research [2], aiSee from AbsInt [1], Tom Sawyer [19], daVinci Presenter by b-novative GmbH [3], and Microsoft Visio [12]. A summary of these tools' information exploration (visualization) capabilities according to the framework is provided in Figure 2.

Static

The dot program is part of the GraphViz suite from AT&T Research. Technically it is a free tool, although the legality of its use in a commercial environment is not obvious at first glance. For this reason, the use of dot by the industrial partner was somewhat limited.

In keeping with the Unix philosophy (where dot was originally developed), dot can be used in a pipeline as a formatting engine similar to the troff text processor. It can also be instructed to use a graph described in a simple text format.

Because of the impressive graph layout capabilities provided by dot, coupled with the ease of generating dot files by third-party reverse engineering tools, dot was selected as the project's candidate program visualization tool in the "static" category.

Interactive

aiSee is based on the graph layout tool VCG (*V*isualization of *C*ompiler *G*raphs*)* that started at Saarland University, Germany in 1991. The goal was to develop a tool that supports the understanding of the internal data structures of a complier. After being redesigned, extended and reimplemented, in January 2000, AbsInt Angewandte Informatik GmbH licensed the non-public version of VCG from the Saarland University for commercial usage.

aiSee can provide 15 basic layouts, for example, fish-eye views can have dynamic interaction. It also supports up to 256 colors, different line shapes and icon shapes. It can also export its graphs to colored Postscript files (on multiple pages for large graphs).

aiSee was selected as the project's candidate program visualization tool in the "interactive" category primarily because of its navigation and subgraph management capabilities. aiSee permits the user to traverse the graph by selecting an edge and then follow the link to the next node. Unfortunately, the interface to permit this important operation is rather non-standard.

aiSee also provides graph folding/unfolding operations. One can switch views between subgraphs' hierarchical view and. detail view. The subgraphs can be folded into a summary node, shown in a box, shown as a cluster, wrapped, or shown exclusively. This feature is useful for showing architecture level of software systems, which was a requirement of the industrial partner.

Editable

The final tool selected was Visio as the project's candidate program visualization tool in the "editable" category. Microsoft's Visio program is part of their successful Office family. Visio is a business and technical drawing and diagramming program. It is designed to convey information visually. It has templates that setup the page appropriately and open stencils that contain pre-drawn shapes.

Visio can be used as both a graphics drawings and a graphics browsing program. It is mostly used to create images such as block diagrams, flowcharts, tables, floor plans, project schedules, and so on. It uses text and symbols to convey information at a glance. Its purpose is not to be a

Activity	Feature	Rigi	Dot	aiSee	Tom Sawyer	daVinci Presenter	Visio
Presentation	Multiple Views	X	X	X	X	X	X
	Visualization Techniques	Multiple, EUP	Hierarchical	Hierarchical	Several	Hier., Tree	Limited
	User Interface	X		Non-standard	Editor Toolkit	Standard Tk	X
Output Formats	GIF, TIFF, JPEG, PNG		X	X		X	X
	PS	X	X	X	X		X
	PDF		X				X
	HTML						X
Batch Mode			X	X		X	X
Quality Attributes	Applicability	Full EUP	Directed, Undirected	Directed		Directed graphs	General Purpose
	Extensibility	Full EUP		X		Tcl/Tk	X
	Scalability	1M Nodes	100K Nodes	100K Nodes	1M Nodes	10K Nodes	100K Nodes
Misc.	Computing Platform	Linux, Windows	Linux, Windows	Solaris, Windows	Windows	Unix, Windows	Windows
	Ancillary Reqs	Tcl/Tk			Layout Toolkit		Office
	Cost	0	0	$$	$$$$$	$$	$$

Figure 2: Information Exploration (Visualization) Tool Evaluation

program visualization tool per se, given its limited layout choices.

Nevertheless, Visio was considered to be a good candidate software visualization tool for the project for several reasons. Firstly, from an adoption point of view, the industrial partner uses Visio in their current develop process. Therefore, they are familiar with its interface and capabilities. Deployment costs would be lowered substantially.

Secondly, from a cognitive support perspective, the graphical documentation produced by Visio can be composed from the set of symbols and icons used by the industrial partner to represent their software and hardware artifacts. These icons come from the real-time design tool they use. The ability to use the same icons to represent recovered artifacts from the industrial partner's source code provides the potential for round-trip reverse engineering.

Finally, our reverse engineering toolset included data gathering engines to extract artifacts from source code and assorted types of documentation. The resultant information is stored in a central database. The database can be queried and updated by visualization tools such as Visio using standard application program interfaces. In this manner, Visio can be used to create diagrams in an automated manner, diagrams that are up to date with respect to the source code of the subject system.

The graph generated by Visio can be saved as a file (static), navigated by the user (with the appropriate Visual Basic code written to provide user interaction), and edited by the user. Therefore, Visio can be considered the most powerful of all the tools considered in our selection process in this specific usage context.

5. SUMMARY

This paper described preliminary work towards a task-oriented classification of program visualization techniques. The classification is currently descriptive in nature, and divides the visualization techniques into three classes (static, interactive, and editable) based on the level of end-user interaction with the generated graphical documentation. The primary advantage of a task-oriented

classification is that it will ultimately map common activities related to program understanding to specific types of software visualization.

To illustrate how the descriptive classification can be used to structure the selection of software visualization tools in support of program understanding, an example from an academic/industrial collaborative project was provided. The specific visualization techniques that were described as most suitable for particular program understanding tasks were based in part on our experiences with the industrial partner. One of the most important next steps is to empirically verify these assumptions through case studies.

There is a hypothesized causal chain between graphical documentation and improved program understanding that this work has begun to evaluate. The next step is to address the specific issue of empirical evidence between different types of graphical documentation and the perceived benefit to program understanding. This will be performed in two phases. In the first phases we will perform a qualitative assessment of the suitability of the three classes of graphical documentation for common program understanding tasks. The second phase is to perform a series of experiments to quantitatively assess the efficacy of these forms of graphical documentation in specific usage contexts in support of program understanding. The result of this research should be a task-oriented classification framework of program visualization techniques.

REFERENCES

[1] AbsInt GmbH. "aiSee – Graph Visualization." Online at www.absint.com/aisee.

[2] AT&T Research. "The Dot Graph Visualization Program." Online at www.graphviz.org.

[3] b-novative GmbH. "daVinci Presenter." Online at www.b-novative.com/products/daVinci/daVinci.html.

[4] Bonetto, T.; Lantrip, D.; Torzewski, S.; and Ramsdale, C. "Emerging Tools for Information Visualization." *IT Professional* 2(6):14-20, November 2000.

[5] Buchsbaum, A.; Chen, Y.-F.; Huang, H.; Koutsofios, E.; Mocenigo, J.; Rogers, A.; Jankowsky, M.; and Mancoridis, S. "Visualizing and Analyzing Software Infrastructures." *IEEE Software* 18(5):62-70, September/October 2001.

[6] Chikofsky, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7(1):13-17, January 1990.

[7] Georgia Institute of Technology. Online at www.cc.gatech.edu/gvu/softviz.

[8] Hartmann, J.; Huang, S.; and Tilley, S. "Documenting Software Systems with Views II: An Integrated Approach Based on XML." *Proceedings of the 19th Annual International Conference on Systems Documentation* (SIGDOC 2001: Santa Fe, NM; October 21-24, 2001), pp. 237-246. ACM Press: New York, NY, 2001.

[9] Herman, I.; Melançon, G.; and Marshall, S. "Graph Visualization and Navigation in Information Visualization: A Survey." *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24-43, 2000.

[10] Knight, C.; Munro, M. "Comprehension with[in] Virtual Environment Visualizations". *Proceedings of the 7th International Workshop on Program Comprehension* (IWPC 1999: May 5-7, 1999; Pittsburgh, PA, USA), pp. 4-11. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[11] Koschke, R. "Software Visualization: Does Anyone Care?" Survey available online at www.bauhaus-stuttgart.de. 2001.

[12] Microsoft Corp. "Visio: The Office Business Diagramming Solution." Online at www.microsoft.com/office/visio.

[13] Shatner, W. (with Reeves-Stevens, J. and Reeves-Stevens, G). *Dark Victory.* New York, NY: Pocket Books, 1999.

[14] Storey, M.-A.; Fracchia, F.; and Müller, H. "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization." *Proceedings of the 5th International Workshop on Program Comprehension* (IWPC 1997: May 28-30, 1997; Dearborn, MI), 17-28. Los Alamitos, CA: IEEE CS Press, 1997.

[15] Storey, M.-A.; Wong, K.; Fracchia, F.; and Müller, H. "On Integrating Visualization Techniques for Effective Software Exploration." *Proceedings of IEEE Symposium on Information Visualization* (InfoVis'97: October 20-21, 1997; Phoenix, AZ), pp. 38-45. Los Alamitos, CA: IEEE CS Press, 1997.

[16] Tilley, S. *A Reverse-Engineering Environment Framework* (CMU/SEI-98-TR-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998.

[17] Tilley, S.; Huang, S. "On Selecting Software Visualization Tools for Program Understanding in an Industrial Context". *Proceedings of the 10th International Workshop on Program Comprehension* (IWPC 2002: June 26-29, 2002; Paris, France), pp. 285-288. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[18] Tilley, S.; Müller, H.; and Orgun, M. "Documenting Software Systems with Views." *Proceedings of the 10th Annual International Conference on Systems Documentation* (SIGDOC '92: Ottawa, ON; October 13-16, 1992), pp. 211-219. ACM Press: New York, NY, 1992.

[19] Tom Sawyer Software, Inc. "Graph Layout Toolkit." Online at www.tomsawyer.com/glt.

[20] W3C. "Scalable Vector Graphics (SVG)". Online at www.w3.org/Graphics/SVG.

[21] Wiki Webs. Online at www.wiki.org.