# Documenting Software Systems with Views VI: Lessons Learned from 15 Years of Research & Practice

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

## ABSTRACT
A "view" is a form of graphical documentation representing some aspect of a software system. Views can be an important aid in helping to understand large-scale applications, and can be automatically produced through reverse engineering. This paper summarizes our findings and lessons learned related to documenting software systems with views from numerous projects spanning 15 years of research and practice (1992-2007). These findings have continued relevancy for modern software development and technical communication alike.

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

## General Terms
Documentation, Human Factors

## Keywords
graphical documentation, views, visualization, XML, UML, program understanding, reverse engineering

## 1. INTRODUCTION
Between 1992 and 2007 there have been five papers published at SIGDOC conferences focused on documenting software systems with views. These papers explored the role of program documentation as an aid in helping a software engineer understand large-scale systems. Such documentation can take many forms (e.g., textual or graphical), can be produced in a variety of ways (e.g., manually by the developer or automatically by a tool), and serve multiple purposes (e.g., internal to the team or external to the user). In this context, a "view" is a specific instance of such program documentation.

From a software architecture perspective, a "view" is defined as a representation of an entire system consisting of a set of related concerns [6]. In other words, a single view describes architecture from the perspective of a particular user (stakeholder), all of whom share a common interest (the subject system). In this context, our definition of "view" is more closely aligned with the notion of a "viewpoint" in the IEEE standard. Minor variations in definitions aside, the essential aspects of a view are the same.

Over the last 15 years we have conducted extensive research into the use of views as a form of graphical documentation. This paper summarizes our findings from this work and the results from numerous view-related projects that we have been involved in. These projects have focused on many different types of software systems, involving different documentation requirements for a wide range of users. This diversity has helped mature the techniques used to produce the views, and has also informed our understanding of the role of graphical documentation in multiple contexts.

Research into views is an interdisciplinary activity, involving software engineering and technical communication in equal measure. As such, the work has given rise to a number of related threads of inquiry, such as the Graphical Documentation (GDOC) workshops that have been held since 2001 [10]. These workshops explored specific aspects of views, such as style guidelines, that further expanded the body of knowledge for the area.

The next three sections provide a detailed summary of the *Views* papers from 1992-2007. Section 2 provides an overview of the work on views of legacy software systems. Section 3 discusses the work on views for Web applications. Section 4 describes how views are applied to visualizing software design patterns. Finally, Section 5 summarizes the paper, discusses some of the lessons learned from these 15 years of research and practice, and outlines possible avenues for future work.

## 2. VIEWS OF LEGACY SYSTEMS
Legacy systems are large-scale, long-lived, complex software applications. They 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.


*SIGDOC'09,* October 5–7, 2009, Bloomington, Indiana, USA.

## 2.1 Views I

The *Views I* paper [11] directly addressed this program understanding challenge. The paper described a reverse engineering methodology for building subsystem structures out of software building blocks, and how documenting a software system with graphical views created by this process can produce numerous benefits. Reverse engineering is the process of extracting system abstractions and design information out of existing software systems for maintenance, re-engineering, and reuse [1]. This process involves identifying software artifacts in a particular representation of a subject system via mental pattern recognition by the reverse engineer, and the aggregation of these artifacts to form more abstract system representations.

An example of the graphical view produced using the proprietary reverse engineering toolset called Rigi [7] is shown in Figure 1. Notable in the figure is the use of non-standard icons to represent software artifacts and unlabeled links to represent their inter-relationships. Since this work pre-dates the use of graphical standards, such as the Unified Modeling Language (UML), or common diagramming tools that provide stencil support such as Microsoft Visio, there was little alternative to using proprietary graphical representations in the views.

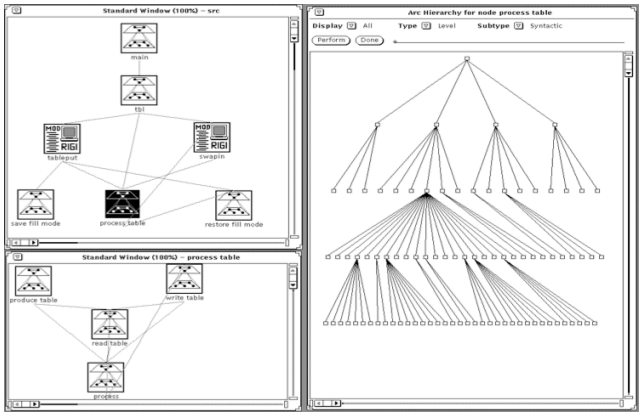


**Figure 1: View of a legacy software system**

The *Views I* paper primarily addressed the needs of the software engineer and technical manager as document users in the context of maintenance of existing software systems. In particular, the technique described in the paper focused on providing three key benefits: (1) aiding management decisions; (2) recovering lost information; and (3) improving system comprehension. By 1992 the technique was shown to be successful when applied to medium-sized systems (up to 100 KLOC); it was subsequently used on much larger systems with similar effectiveness.

## 2.2 Views II

The *Views II* paper [4] further developed this theme. The focus remained on using reverse engineering technology as an aid in program understanding, specifically tailored to address what were seen as three prominent shortcomings of existing documentation techniques: (1) low quality; (2) single perspective; and (3) lack of integration. By 2001 the proprietary toolset and data formats had been replaced by (emerging) industry standards based on XML.

The paper focused on the creation and use of specific Document Type Definitions (DTD) that are defined by MSR as a standard for software documentation. MSR is a consortium of several German automotive companies whose goal is to support cooperative development between car manufacturers and their electronic system suppliers. To illustrate the approach, selected aspects of the document creation process for an engine control system were presented. The XML documentation produced by this process, as viewed by a commercial editor called XMetal, is shown in Figure 2.

The integrated approach to program documentation described in the *Views II* paper offered several benefits over the traditional techniques and directly addressed some of the shortcomings outlined above. For example, the views produced were of high quality in the sense that they were always up-to-date with respect to the underlying source code, in part because the reverse engineering process was semi-automated. The integrated approach also provided multiple, user-selectable (and user-defined) views of the underlying information through both programmatic and graphical interfaces. Lastly, the use of standard XML and commonly-available tools eased the integration of the view creation process into existing software engineering activities, thereby facilitating adoption of the approach by industrial partners and other interested parties – a problem that was encountered in previous attempts at automated document creation.

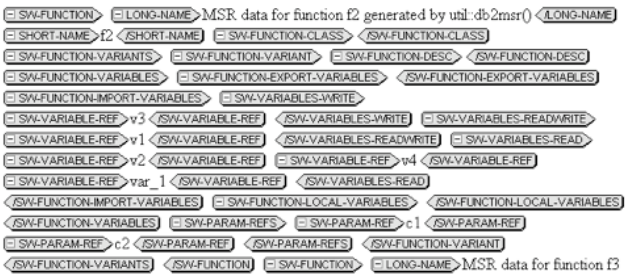


**Figure 2: View of XML-formatted documentation**

## 2.3 Views III

As the role of graphical documentation in aiding program understanding grew more prevalent, a number of questions arose as to their efficacy. It was long assumed that graphical forms of documentation that rely on software visualization techniques do make complicated information easier to understand. 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. In 2002 (and still today) there were many unresolved research issues related to which types of diagrams are most appropriate for aiding program understanding. More specifically, it is unknown exactly which views are most suitable for which types of program understanding tasks, and in which specific usage contexts.

The *Views III* paper [8] described preliminary work towards a task-oriented classification of program visualization techniques. The classification was descriptive in nature, and divided the visualization techniques into three categories (static, interactive, and editable) based on the level of end-user interaction with the generated graphical documentation. In terms of the tools that produce these different visualizations, 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.

Static graphs are the oldest form of graphical documentation. Because static visualizations are hardcopy-like images that the user can view or print, they are relatively easy to produce and to integrate into current software engineering processes. As shown in Figure 3, 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 with the static image; it is included solely as read-only graphical documentation.

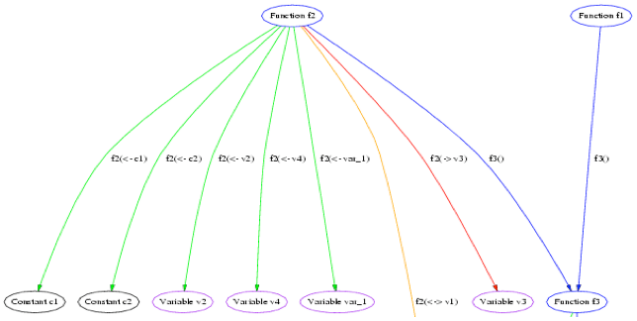


**Figure 3: A static view (call graph)**

The second category of graphical documentation is interactive, which permits the user to navigate the view 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.

The third category of graphs produced by software visualization tools is editable. Editable views let the user actually change the generated graph itself, such as adding new nodes or edges. If the editable view is connected to a backend database, changes 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.

The primary advantage of a task-oriented classification is that it can map common activities related to program understanding to specific types of software visualization. The *Views III* paper also provided 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. This classification was subsequently used in work related to on a documentation maturity model (DMM) [5].

# 3. VIEWS OF WEB APPLICATIONS

Over time the nature and distinguishing characteristics of a legacy system have changed. For example, the first three *Views* papers addressed documenting traditional legacy systems: monolithic, homogenous, and typically written in third generation procedural programming languages. By 2004 a new type of legacy system had developed: Web applications. This new class of legacy system shared many of the negative features of traditional legacy systems (e.g., poor structure, little or no documentation), but they also introduced several new challenges as well. One such new challenge was documenting Web application transactions.

A Web transaction is a collection of serial and/or parallel activities that contributes to achieving a user-oriented business objective using a Web-based application. From a software engineering point of view, neglecting or improperly addressing Web transaction design documentation causes numerous problems that can arise during long-term system evolution. These problems include: (1) difficulties in communication and understanding between designer and customer (first), and designer and developer (later); (2) unpredictability of the development process and resultant application quality; and (3) difficulties in verifying the traceability between business process requirements and transaction implementation.

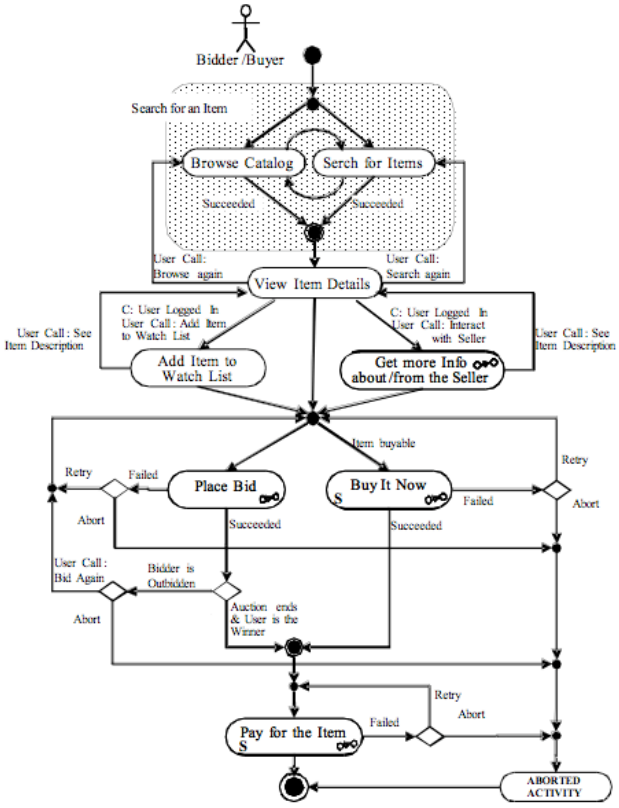


**Figure 4: A view of a Web application execution model**

The *Views IV* paper [2] detailed an approach to documenting the conceptual design of Web transactions. A series of (extended)

UML diagrams were used to graphically document the formalized meta-model, which greatly facilitated adoption of the approach by practicing software engineers and technical writers. Note that standardized graphical representations based on UML had supplanted the proprietary icons and unlabeled links from the *Views I* paper of 1992.

The approach described in the *Views IV* paper also relied on an extension to a formal framework designing ubiquitous Web applications. The meta-model had three sub-models: (1) the organization model (a customized version of a UML class diagram that describes the transaction from a static point of view in terms of the user activities it includes and the hierarchical and semantic relations among these activities.); (2) the execution model (a customized version of a UML activity diagram that models the transaction from a dynamic point of view); and (3) the navigation model (a model that specifically describes the informative aspects of each activity defined by the two previous models and the interplay between informative navigation and execution of the Web application transaction.) An example of the execution model for part of a transaction for "Buy/Bid for an item" from the eBay Web site rendered as a form of graphical documentation is shown in Figure 4.

# 4. VIEWS OF DESIGN PATTERNS

There is a tendency for an area of inquiry to change as it matures. For example, the community may desire more empirical evidence as to the effectiveness of a proposed technique in comparison to existing techniques. There is also a trend towards formalization and codification of best practice. The previous *Views* papers reflect this maturation: from prototype toolsets, non-standard data formats, and proprietary graphical icons and links, to commercial toolsets, standardized data formats (XML), and standardized graphical representations (UML).

Software design has followed a similar evolutionary path of maturation. There are now collections of "design patterns" [3] available to software engineers to inform their design choices. A design pattern is a cooperative assembly of classes that collectively implement the solution to a low-level design problem in a well-known and language-neutral way. They are in effect a codification of best practice in low-level software design.

The reverse engineering techniques described in previous sections of this paper produce graphical documentation artifacts representing a software system. In such redocumentation information products, a software system's design is generally explicated at several levels of detail, including the overall system, its component functional subsystems, and the source code itself. At an intermediate level, between the functional subsystem and source code, object-oriented software can be described in terms of design patterns. Recognizing an implementation of a familiar design pattern might help programmers to more quickly and accurately acquire an understanding of the application. The recognized design pattern might also become a useful operand in further cognitive operations that the programmer performs to make correct decisions during system maintenance. If either of these statements is true, treatment of implemented design patterns adds value to redocumentation as realized in views.

As mentioned earlier, one of the most common strategies for documenting a system's static structure is with UML class diagrams. Similar diagrams are also conventionally used to catalog design patterns. The *Views V* paper [12] argued that system redocumentation in which UML class diagrams are organized primarily for the optimal display of design patterns has several distinct advantages: (1) such documentation fosters program understanding by explicitly exposing implemented design patterns; (2) organization of UML class diagrams around design patterns using the visual design strategies proposed makes UML class diagrams semantically richer and more readable than traditional, comprehensive UML class diagrams; and (3) a reverse engineering tool can semi-automatically generate the described UML class diagrams with minimal resort on the part of the documenting software engineer to a graphical UML design utility.

The paper discussed the challenges inherent in visualizing a software system as a set of design patterns, reviewed the impact of related work, and described a UML-compliant enhanced class-participation diagram as one possible solution. Design patterns have been shown to be useful in conveying design intent, but like all visual forms of graphical documentation they have inherent limitations when it comes to scalability and usability for complex systems [13]. One approach to reducing the visual clutter are grouping strategies that organize a visual field into units and subunits. In essence, grouping strategies facilitate navigability and further cognitive chunking.

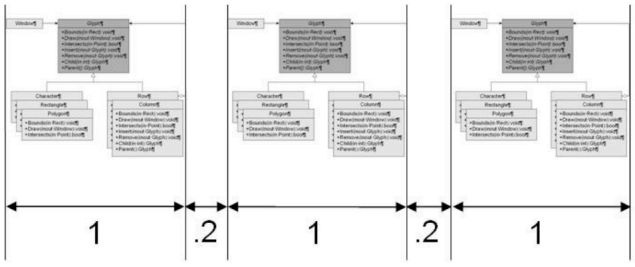


**Figure 5: View of spacing between patterns**

Several grouping strategies were employed in the *Views V* paper, such as spatial nearness grouping as shown in Figure 5. For example, the primary application of spatial nearness grouping in a class-participation diagram is accomplished by the canonical representations of the design patterns themselves, which keep the participating classes in close spatial proximity. Adequate white space margins around each pattern (about 20% of the width of the canonical representations in Figure 5 maintain a distinct boundary between the patterns so that even at a cursory glance it is obvious in which pattern any depicted class is a participant.

# 5. SUMMARY

This paper summarized 15 years of research into the use of views to graphical document software systems as an aid to program understanding. From 1992 to 2007 there have been five papers published at SIGDOC conferences on this work. The *Views I* paper described a reverse engineering methodology for building subsystem structures out of software building blocks, and how documenting a software system with graphical views created by

this process can produce numerous benefits. The next two *Views* papers further developed this theme. In contrast, the *Views IV* paper described an approach to documenting the conceptual design of Web transactions. The most recent work in the area was summarized in the *Views V* paper, which discussed the challenges inherent in visualizing a software system as a set of design patterns represented as styled UML diagrams.

Although the nature of the software systems under examination has changed dramatically during this time, the fundamental findings from the *Views* work have continued relevancy for modern software engineering practice and technical communication studies.

## 5.1 Lessons Learned

As can be seen from the differing nature of the *Views* papers between 1992 and 2007, with the change from monolithic software systems to Web application transactions to sophisticated use of design patterns, there is increasing complexity and therefore increasing need to provide effective forms of documentation. The applications may be different, but in the 15 years of work in this area, we have learned numerous lessons, from both research and practical perspectives, that cut across all of our studies. Three of the most important lessons learned are: (1) to constantly question the efficacy of graphical documentation; (2) the use of standards aids the adoption of views produced through automated tools; and (3) the importance of a technical communicator in shaping the use of views for maximum usefulness is growing as the complexity of the software systems also grows.

### *5.1.1 Assessing Efficacy*

Significant research goes into creating sophisticated visualizations of software systems, under the assumption that eye-pleasing graphic representations are the preferred format for program documentation, but our studies have found that this is not always true. In other words, the question of when graphical documentation is more effective than other forms of documentation (e.g., textual), and for which types of users, remains open.

Although the *Views* research has attempted to address this question through the classification scheme for visualizations described in Section 2.3, and in related work focused on empirical studies assessing the efficacy of different types of UML diagrams [9], many issues are unresolved. A lesson learned from this experience might be stated as: "graphics are useful – except when they're not." It's distinguishing these different contexts that has proven difficult.

### *5.1.2 Leveraging Standards*

The first *Views* papers described a reverse engineering technique that relied on proprietary tools and non-standard glyphs (icons and links). This limited interoperability with other tools. It also made adoption of the approach more difficult for everyday users. Over time, commercial tools, standard data formats (e.g., XML), and standardized graphical representations (e.g., UML) were used in the *Views* projects. This improved the receptiveness of users to the automated production of views. But it also somewhat limited innovation by forcing the use of certain representations (e.g., UML class diagrams) when they weren't perfect matches for the graphical task at hand.

A lesson learned from this experience might be stated as: "standards are good – except when they're not." The challenge is finding the right balance between what might be a superior graphical representation of a concept versus what the user is accustomed to seeing on their screens.

### *5.1.3 The Role of Technical Communication*

It is rarely the case that software systems grow easier to understand over time. Rather, they grow increasingly complex, which in turn makes them harder to understand. The proliferation of new technologies, programming languages, and infrastructure platforms only adds to the problem. Views can help one understand the system, but they are more effective if they adhere to commonly accepted best practice in terms of issues such as layout, style, and proven tenets of good communication.

The problem is that technical communicators are not often involved in this process. The result is that engineers, who have the best of intentions but who lack the background necessary to fully exploit the capabilities of graphical documentation to support the end users' tasks, create the views without the benefit of a corpus of experience. A lesson learned from this experience might be stated as: "we need to talk." There is a clear need for education in both directions: between the software engineers creating the tools, and the technical communicators who have the knowledge and training to guide the creation of views to maximal efficacy.

## 5.2 Future Work

Much of our current work focuses on addressing the issues raised in the "Lessons Learned" described above. There is a clear need for more empirical situated studies for views. Coming to an agreement on the use of appropriate standards across tools – particularly for graphical representations of software artifacts between UML diagrams – would be beneficial. Finally, continued education and information exchange between the view creators and view users will serve to improve the final product and usage experience for all.

## ACKNOWLEDGEMENTS

Many thanks to the co-authors and collaborators who have worked on the *Views* research project over the years. These include (in alphabetical order) Damiano Distante, Jochen Hartmann, Shihong Huang, Hausi Müller, Mehmet Orgun, and Tim Trese. Without their insightful guidance and support much of this work would not have been possible.

## REFERENCES

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

[2] Distante, D.; Tilley, S.; and Huang, S. "Documenting Software Systems with Views IV: Documenting Web Transaction Design with UWAT+." *Proceedings of the 22nd International Conference on Design of Communication* (SIGDOC 2004: October 10-13, 2004; Memphis, TN), pp. 33-40. ACM Press: New York, NY, 2004.

[3] Gamma, E.; Helm, R.; Johnson, J.; and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[4] 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.

[5] Huang, S. and Tilley, S. "Towards a Documentation Maturity Model." *Proceedings of the 21st Annual International Conference on Design of Communication* (SIGDOC 2003: October 12-15, 2003; San Francisco, CA), pp. 93-99. ACM Press: New York, NY, 2003.

[6] IEEE P1471. "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems." Standard 1471-2000.

[7] Kienle H.; and Müller, H. "The Rigi Reverse Engineering Environment." *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques* (WASDeTT 2008: July 8, 2008; Paphos, Cyprus). Co-Located with ECOOP 2008.

[8] Tilley, S and Huang, S. "Documenting Software Systems with Views III: Towards a Task-Oriented Classification of Program Visualization Techniques". *Proceedings of the 20th Annual International Conference on Systems Documentation* (SIGDOC 2002: October 20-23, 2002; Toronto, Canada), pp. 226-233. ACM Press: New York, NY, 2002.

[9] Tilley, S. and Huang, S. "A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding." *Proceedings of the 21st Annual International Conference on Design of Communication* (SIGDOC 2003: October 12-15, 2003; San Francisco, CA), pp. 184-191. ACM Press: New York, NY, 2003.

[10] Tilley, S.; and Bellomo, S. "7th International Workshop on Graphical Documentation: Documenting SOA-Based Systems." To be held as part of the *27th ACM International Conference on Design of Communication* (SIGDOC 2009: Oct. 5-7, 2009; Bloomington, IN).

[11] 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. New York, NY: Association for Computing Machinery, 1992.

[12] Trese, T. and Tilley, S. "Documenting Software Systems with Views V: Towards Visual Documentation of Design Patterns as an Aid to Program Understanding." *Proceedings of the 25th ACM International Conference on Design of Communication* (SIGDOC 2007: Oct. 22-24, 2007; El Paso, TX), pp. 103-112. ACM Press: New York, NY, 2007.

[13] Tufte, E. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Cheshire, CT: Graphics Press, 1997.