Documenting-in-the-large vs. Documenting-in-the-small[†]

Scott R. Tilley

Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
Tel: (604) 721-7294, Fax: (604) 721-7292
E-mail: `stilley@csr.uvic.ca`

Abstract

There is a significant difference between documenting large programs and documenting small ones. By large programs we mean on the order of 1,000,000 lines, usually written by many different people over a long period of time. Most software documentation may be termed *documenting-in-the-small*, since it typically describes the program at the algorithm and data structure level. To understand large legacy systems, one needs *documenting-in-the-large*: documentation describing the high-level structural aspects of the software system's architecture from multiple perspectives. This paper outlines an approach to supporting software evolution through documenting-in-the-large. The approach is based on a flexible reverse engineering process which uses *virtual subsystem stratifications* to represent multiple abstract views of a software system.

Keywords: documenting-in-the-large, reverse engineering, software evolution

[†]This work was supported in part by the British Columbia Advanced Systems Institute, IBM Canada Ltd., the IRIS Federal Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, the Science Council of British Columbia, and the University of Victoria.

Reprinted from *Proceedings of CASCON '93*, (Toronto, Ontario; October 25-28, 1993), pages 1083-1090.

1 Introduction

There are significant differences between software systems of 1,000 lines, of 100,000 lines, and of 1,000,000 lines. One of the most important of these differences is the use of documentation as an aid in program understanding. Most software documentation is *documenting-in-the-small*, since it typically describes the program at the algorithm and data structure level. For large legacy systems, an understanding of the structural aspects of the system's architecture is more important than any single algorithmic component.

This is especially true for software engineers and technical managers responsible for the maintenance of such systems. The documentation that exists for these systems typically describes isolated parts of the system; it does not describe the overall architecture. Moreover, the documentation is often scattered throughout the system and on different media. It is left to maintenance personnel to explore the low-level source code and piece together disparate information to form high-level structure models. Manually creating one such architectural structure document is a difficult task; creating multiple documents that describe the architecture from multiple viewpoints would seem unlikely. Yet it is exactly this sort of *documenting-in-the-large* that is needed to expose the structure of large software systems. One way of producing such documentation for an existing software system is through *reverse engineering.*

Reverse engineering is the process of extracting system abstractions and design information from existing software systems. This information can then be used for subsequent development, maintenance, re-engineering, and project management purposes. The process involves the identification of software artifacts in the subject system, and the organizing of these artifacts to form more abstract system representations and reduce complexity. Through reverse engineering, the overall structure of the subject system can be determined and some of its architectural design information recovered.

Software structure refers to a collection of artifacts that software engineers use to form mental models when designing, documenting, implementing, integrating, inspecting, or analyzing software systems. Artifacts include software *components* such as procedures, modules, subsystems, and interfaces; *dependencies* among components such as supplier-client, composition, and control-flow relations; and *attributes* such as component type, interface size, and interconnection strength. The structure of the system is the organization and interaction of these artifacts [1].

This paper comments on the deficiencies in traditional documentation techniques and describes an approach to supporting software evolution through documenting software structure via reverse engineering. In particular, it focuses on how the approach facilitates understanding the structure of large software systems through the reconstruction and redocumentation of the subject system's *operational architecture*. The structure is created semi-automatically by a software engineer aided by a flexible reverse engineering environment, is modeled using a variety of *software interconnection models*, and is represented by *virtual subsystem stratifications*: multiple abstract views of the software system.

The next section discusses the challenges presented by software evolution. Section 3 outlines the shortcomings of traditional documentation techniques when applied to legacy software systems. Section 4 presents an approach to documenting software structure. Finally, Section 5 summarizes the paper.

2 Software evolution

The primary business of software used to be new development; now it is maintenance [2]. This shift was inevitable: the software profession has reached a turning point, one where more people are employed to maintain existing applications than to develop new systems from scratch. The key phrase is "from scratch." Very little development is done from scratch in other engineering disciplines; so-called new development is done by building upon the results of others using reusable building-blocks (for example, integrated circuits in electronics).

The increasingly high profile of software maintenance has led to *software evolution* being identified as a central software engineering problem [3]. Yet software evolution remains a difficult aspect of the software process. Even the term itself is somewhat ill defined. Traditional approaches to the software process have placed too much emphasis on the artificial distinction between development and maintenance. Evolution begins early in the development process, and the distinction between development and maintenance should be abandoned in favor of an evolutionary process [4].

The implication of this shift in focus, from developing new systems to maintaining and upgrading existing ones, means one must be able to understand the design of the system before one can build on top of it. While design may be difficult [5], reconstructing and effectively (re)documenting the design of existing systems is even more difficult. Recognizing abstractions in real-world systems is as crucial as designing adequate abstractions for new systems.

The hardest task of maintenance is understanding the structure of the system. This is exceedingly difficult for large legacy systems and requires a different approach to program understanding than has been used for programming-in-the-small [6]. The difference in scale from systems of 1,000 lines to 1,000,000 lines cannot be over-emphasized. In the million-lines-of-code range, we need tools to help us *read* programs; text (code) by itself is not very helpful. By "read" one does not mean interpreting each line of source code like a human compiler. Rather, tools should help us "read" the high-level design inherent in the architecture, to gain an understanding of the *gestalt* of the entire system. Documentation has traditionally served an important role in this regard. Yet as the next section describes, traditional approaches to program documentation do not scale up well.

3 Documenting software

The importance of high-quality documentation in program understanding is widely recognized [7]. Without it, the only source of reliable information is the source code itself [8]. While architectural rediscovery may not be a problem for a single developer,[1] or even for a small team (while they are together), it is a problem for long-term large-system evolution. Software engineers and technical managers base many of their project-related decisions on their understanding of the architecture of the software systems they are responsible for. While they rely on original design documents, maintenance histories, and experienced project members (if they are available) to help them understand how a program works, internal documentation is often their primary source of information. Hence, the most obvious way to support program comprehension is to produce and maintain adequate documentation [9].

This section describes three deficiencies of traditional documentation techniques and introduces documenting-in-the-large as an alternative for documenting legacy software systems.

3.1 Documenting-in-the-small

Documentation techniques have not really changed very much in thirty years; most software documentation is still targeted towards in-the-small activities. Documenting isolated algorithms and data structures is a useful way to gain local understanding, and one-paragraph headers at the top of source files, which attempt to describe the main purpose of the module, are invariably read by maintainers. However, for large system evolution, this type of documentation rarely addresses the need of the software engineer to gain an overview of the entire system or of selected subsystems. It is left to the documentation's user to piece together these low-level descriptions to form high-level abstractions.

[1]Even this point could be debated. If the program is sufficiently large or complex, a programmer may have trouble understanding code written just months ago. Consequently, his or her mental model of the system's structure becomes fuzzy at best.

The system-level in-the-large documentation that does survive for legacy systems was probably written during the software's initial design; rarely does it accurately reflect the current implementation. As the software evolves, the design document is left untouched while the implementation drifts farther and farther away from the original designer's intent.

Creation of in-the-small documentation is usually left to the individual developers and maintainers. Unfortunately, documenting software rarely ranks high on their list of activities [10]. It is often considered something to be put off until the last possible moment; in many cases, it is postponed indefinitely. While there do exist model programmers who carefully document design and implementation decisions while working on a piece of code, usually documentation is tacked on as an afterthought. Maintenance is often crisis-driven [11]; pressures of day-to-day development seem to take precedence over documentation. As changes to the program continue, the original documentation becomes increasingly out-of-date.

Even if the documentation is created and maintained, it provides just a single perspective: that of its author. Although each person may have different objectives, everyone will see the same thing: in-line and block commentary with the source code, and original design documents and maintenance logs.[2] Hence, such documentation is not well-suited to serve its many different readers, all of whom should be able to use the same underlying information—at different levels of detail. For example, different levels of documentation are required for the casual user of a program, for the developer familiar with the code, for the maintainer unfamiliar with the system, for testers and technical writers trying to understand its functionality, and for project management personnel looking for "the big picture": an external view of the system's architecture and history [12, 13, 14]. It would be far better to be able to provide documents that describe system architecture from multiple perspectives. In this way, each user would be provided with views of the program that best suited his or her needs.

Thus, traditional approaches to software

[2]Assuming these documents exist.

documentation suffer from at least three major flaws. The documentation produced is: (1) in-the-small; (2) usually out-of-date; and (3) provides just one perspective.

3.2 Documenting-in-the-large

Legacy software systems are too large and ill-structured to be solved by current documentation techniques. Understanding these systems involves uncovering the system-level design of software: discovering the kinds of modules and subsystems used and the way these modules and subsystems are organized. This level of abstraction is called the software architecture level [15, 16].

Managing complexity and supporting evolution are two fundamental problems with large-scale software systems [17]. As the size of software systems increases, the design problems go beyond the algorithms and data structures of computation [18]. Without proper documentation or aid, gaining an appreciation of the overall structure of such systems is a daunting task. Yet it is essential that one understand the structure of a large system to work with it effectively. At this level, the objects of interest are subsystems and their interactions. Before looking at details, one looks for global structure: what modules the system comprises, how are they organized, and how they interact.

Although the proper use of information hiding and separation of concerns can make system structure simpler, it is not a panacea. It leads to a proliferation of small parts, so much so that it is difficult to understand their interrelationships [19]. Since good software engineering design suggests that modules be kept relatively small, the number of modules in a large system is significant [20]. For example, in a system of 500,000 lines, with roughly 200 lines per module, there would be 2,500 modules. This is an order of magnitude more than there are lines of code in each module. It would seem clear that we need a way of documenting this *meta-modularization* of modules into subsystems.

As a result of this structural complexity, the majority of the errors introduced during maintenance are a result of inadequate understanding on the part of the programmer of how his or her modification affects the rest of the system. The clearer and more accurate the mental mode of the system and how the piece fits into the whole, the less likely are errors of this kind to occur. Therefore, the accuracy of the programmer's mental model is of critical importance. Program understanding is the process of acquiring and updating this model. The problems can be ameliorated by documentation that makes the global structure visible and thus can impart an understanding of the overall structure. Global structural visibility of large systems is essential to our ability to understand them. However, documenting structure at the global level alone is not sufficient. A system decomposition must satisfy local readability and global structural visibility if it is to promote understanding [21].

Most human beings visualize structure graphically. Software designers often describe the architecture of particular systems using block diagrams of the major system components and labels that refer to their major functions. Modern interactive systems with graphical display capabilities facilitate the direct manipulation, processing, and presentation of information in graphical form. The use of textual representations is still the predominant form of programming; the use of visual programming languages for programming-in-the-large is relatively new [22]. While a program is logically a hierarchical structure, the program source is physically flat. Compilers are adept at reconstructing the syntactic hierarchy; humans have the much more difficult task of reconstructing the logical hierarchy. This poor representation of programs is a hindrance to program understanding.

For documentation purposes, one is often forced to convert diagrams of system structure to a textual form for computer processing.[3] The textual representation then becomes the primary one and the diagrams frequently become obsolete and ignored. Modern systems make it possible for such diagrams to become an integral part of the systems they represent. It would seem that the traditional approach to

[3] For example, module interconnection languages such as NuMIL [23] are often used to represent module structure and interactions.

the problem is now outdated; switching to textual form is no longer necessary.

4 Documenting software structure

Section 2 shows clearly that any response to the software evolution challenge must address the problems of effectively documenting software structure. The proposed solution to these problems is based on providing a flexible method of identifying, documenting, representing, and presenting the structural aspects of software architecture through reverse engineering. This entire process is called *redocumenting-in-the-large*.

Parnas coined the term "design through documentation" [24]. Our approach is to make the documentation a "live" representation of the source code, not separate text. If the documentation *is* the structure, and vice versa, no discrepancy exists [25]. The rest of this section describes the basis for documenting software structure: virtual subsystem stratifications (VSS's) and software interconnection models (IM's).

4.1 VSS's

Classical architecture has concepts that are desirable for flexible software architecture, including multiple views and architectural styles. For example, a building architect would provide one representation of the building to the carpenter, perhaps another to the plumber, and yet another to the buyer. For software, we presently do with just one view: the implementation. Keeping with the building analogy, this is like a building with no outer skin and all the details exposed: it makes understanding of the overall architecture very difficult.

A virtual architecture imposes a logical structure on a physical system. Limiting modularizations to those supported by file systems and programming languages is not sufficient to support the multiple representations desired for documenting-in-the-large. Instead, virtual modularizations impose logical clustering on user-defined artifacts (subsystems). Stratification means to divide into groups and/or to form into layers. It is through stratification and virtual modularization that one represents two dimensions of software structure: inter-layer and intra-layer dependencies, respectively.

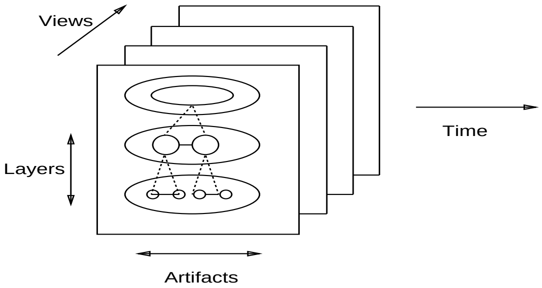

Figure 1: Evolutionary VSS's

It would seem that just these two layers are not enough, however. A third dimension is required to represent multiple views [26]. One then can represent a software system as a cube, with each two-dimensional "slice" a particular stratification. Taken together, these slices form a complete set of documentation for a software system—for one release. To support multiple releases and change information, a fourth dimension may be used. A four-dimensional hypercube is the result: a three-dimensional cube moving through time. A depiction of this concept is presented in Figure 1.

Hierarchical subsystem structures are created through the reverse engineering process described in [27]. They can be used to impose logical structure on legacy systems. Since prolonged maintenance tends to degrade software structure, it is sometimes advantageous to disregard the existing modularization based on the source code's physical structure. Instead, one can construct aggregations of software artifacts based on whatever clustering and selection criteria are deemed appropriate for enhancing the understanding of the entire system or of selected subsystems of relevance. Different views of the software system may be produced for diverse audiences by using different clustering guidelines. Moreover, these views may coexist simultaneously and are kept up-to-date automatically [28].

It is important in program understanding to construct program representations that involve

$\textsf{Intra-layer IM} = (\{\textit{artifact}\}, \{\textit{depends-on}\})$
$\textsf{Inter-layer IM} = (\{\textit{subsystem}\}, \{\textit{is-part-of}\})$
$\textsf{Inter-release IM} = (\{\textit{predicates}\}, \{\textit{satisfies}\})$

Figure 2: Sample interconnection models

concepts from the application domain. Often, they will not be directly represented in the code and may only be known informally by the maintainer [29]. Virtual subsystems can be used to represent such implicit mappings. They can be used to understand and describe existing software systems for risk analysis and project management purposes [30].

4.2 IM's

Interconnection models are a formalism used to describe relationships among objects in a software system as a set of tuples [31]:

$$\textsf{IM} = (\{\textit{objects}\}, \{\textit{relationships}\})\ .$$

The granularity of the inter-object relationships depends on the programming language, the support environment, and the visibility control mechanisms used [32].

This set of tuples conveniently maps to a graph structure, with the objects being nodes and the relationships being attributed arcs between the nodes. As a formalism, IM's also have the advantage of allowing structural queries to be performed on the VSS's. IM's can be generalized to represent arbitrary interconnections among objects. For example, a novel interpretation of the visual and spatial relationships among objects and images in a graphical system is described in [33].

There are several other such useful interconnection models that are applicable to program understanding. For example, intra-layer dependencies are well-modeled by relationships such as the unit and syntactic interconnection model. Inter-layer dependencies define how one layer is related to another. Examples include composition, nesting, and inheritance. Dependencies between structural alternatives may be represented by maps. Finally, dependencies that support inter-release structural impact analysis may be modeled by variants of the semantic interconnection model. Examples of these are shown in Figure 2.

5 Summary

Legacy software systems require a different approach to software documentation than has traditionally been used. As an aid in program understanding for large, evolving software systems, documenting software structure plays a key role. One way of producing accurate in-the-large documentation is through reverse engineering. Through this process, one is able to to produce accurate "operational" design documents describing the architecture of the software system's current state—not that of the original system before numerous maintenance changes were made.

An approach to supporting software evolution by identifying, documenting, representing, and presenting the structural aspects of a system's architecture was presented. The approach makes use of virtual subsystem stratifications representing multiple abstract views of a software system. It addresses the three problems of traditional in-the-small documentation techniques outlined in Section 3.

A unique aspect of this approach is that the structural documentation produced is always up-to-date, since it is based on the underlying source code. Moreover, the structural redocumentation does not involve physically restructuring the code (although this might be a desirable outcome); the documentation produced represents virtual (re)modularizations at various levels of detail.

About the author

Scott R. Tilley is currently on leave from IBM Canada Ltd., pursuing a Ph.D. in the Department of Computer Science at the University of Victoria. His field of research is software engineering in general, and program understanding, software maintenance, and reverse engineering in particular. He can be reached at the University of Victoria, or at the IBM PRGS Toronto Laboratory, 844 Don Mills Rd., 22/121/844/TOR, North York, ON, Canada M3C 1V7. His e-mail ad-

dresses are `stilley@csr.uvic.ca` at UVic, and `stilley@vnet.ibm.com` or `TOROLAB6(TILLEY)` (VNet) at IBM.

References

[1] H. L. Ossher. A mechanism for specifying the structure of large, layered systems. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, 1987.

[2] R. L. Glass. We have lost our way. *The Journal of Systems and Software*, 18(3):111–112, March 1992.

[3] W. F. Tichy, N. Habermann, and L. Prechelt. Summary of the Dagstuhl workshop on future directions in software engineering. *ACM SIGSOFT Software Engineering Notes*, 18(1):35–48, January 1993.

[4] D. C. Poo and P. J. Layzell. An evolutionary structural model for software maintenance. *The Journal of Systems and Software*, 18(2):113–123, May 1992.

[5] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[6] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[7] J. E. Huffman and C. G. Burgess. Partially automated in-line documentation (PAID): Design and implementation of a software maintenance tool. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 60–65. IEEE Computer Society Press (Order Number 879), October 1988.

[8] N. T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 54–59. IEEE Computer Society Press (Order Number 879), October 1988.

[9] J. Sametinger. A tool for the maintenance of C++ programs. In *CSM'90: Proceedings of the 1990 Conference on Software Maintenance,* (San Diego, California; November 26-29, 1990), pages 54–59. IEEE Computer Society Press (Order Number 2091), November 1990.

[10] L. Landis, P. Hyland, A. Gilbert, and A. Fine. Documentation in a software maintenance environment. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 66–73. IEEE Computer Society Press (Order Number 879), October 1988.

[11] S. Paul, A. Prakash, E. Buss, and J. Henshaw. Theories and techniques of program understanding. In *Proceedings of CASCON'91,* (Toronto, Ontario; October 28-30, 1991), pages 37–53. IBM Canada Ltd., October 1991.

[12] F. P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, 1982.

[13] P. J. Brown. Interactive documentation. *Software — Practice and Experience*, 16(3), March 1986.

[14] D. Ressler and D. Stribling. Designing and prototyping a portable hypertext application. In *SIGDOC'90 Conference Proceedings*, pages 88–94, October 1990.

[15] M. Shaw. Larger scale systems require higher-level abstractions. *ACM SIGSOFT Software Engineering Notes*, 14(3):143–146, May 1989. Proceedings of the Fifth International Workshop on Software Specification and Design.

[16] T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute; Carnegie-Mellon University, November 1990.

[17] F. P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[18] M. Shaw and D. Garlan. Experiences with a course on architectures for software systems. Technical Report CMU-CS-92-176, Carnegie-Mellon University, 1992.

[19] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–137, March 1979.

[20] Z. L. Lichtman. Generation and consistency checking of design and program structures. *IEEE Transactions on Software Engineering*, SE-12(1):172–181, January 1986.

[21] H. L. Ossher. A case study in structure specification: A grid description of Scribe. *IEEE Transactions on Software Engineering*, 15(11):1397–1416, November 1989.

[22] D. A. Penny. *The Software Landscape: A Visual Formalism for Programming-in-the-Large*. PhD thesis, The University of Toronto, November 1992.

[23] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, January 1990.

[24] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[25] B. Kernighan and P. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1974.

[26] H. L. Ossher. *A New Program Structuring Mechanism Based on Layered Graphs*. PhD thesis, Stanford University, 1984.

[27] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 1993. In press.

[28] K. Wong. Managing views in a program understanding tool. In the Proceedings of *CASCON '93,* (Toronto, Ontario; October 25-28, 1993), pages 244–249, October 1993.

[29] W. Howden and S. Pak. Problem domain, structural and logical abstactions in reverse engineering. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Orlando, Florida; November 9-12, 1992), pages 214–224. IEEE Computer Society Press (Order Number 879), November 1992.

[30] S. R. Tilley and H. A. Müller. Using virtual subsystems in project management. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 144–153, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[31] D. E. Perry. Software interconnection models. In *ICSE'9: Proceedings of the 9th International Conference on Software Engineering,* pages 69–69, April 1987.

[32] A. L. Wolf, L. A. Clarke, and J. C. Wileden. A model of visibility control. *IEEE Transactions on Software Engineering*, 14(4):512–520, April 1988.

[33] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments,* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992. In *ACM Software Engineering Notes*, 17(5).