The Early History of Software Visualization

Ronald Baecker
University of Toronto

This chapter presents the early history of software visualization. It positions the field as a branch of software engineering that strives to aid programmers in managing the complexity of modern software. Unfortunately, as systems such as Windows 95 contain over 10,000,000 lines of code, software visualization has a long way to go if it is to play a substantial role where the need is greatest.

Managing Complex Software

A *program* is a precise description, expressed in a *computer programming language*, of a system, process, or problem solution. Large programs typically progress through a *life cycle* (Belady and Lehman, 1976) which includes *debugging*. They are refined and often redesigned and reimplemented as part of an iterative, user-centred design approach (Baecker, Grudin, Buxton, and Greenberg, 1995) involving interactions with and feedback from users. Long-term use requires that *maintenance* be done throughout the program's lifetime. Maintenance often consumes 50% to 75% of the total costs incurred over that lifetime (Boehm, 1981, p. 533).

Software creation and maintenance is difficult and costly because most real programs are complex and hard to understand. Reasons for this include:

• We increasingly demand more and more functionality in programs and in *systems* of programs, therefore often requiring millions of lines of code.
• The specifications of large programs continually evolve as they are used. Systems must frequently be modified to meet these changing specifications.
• Turnover in the software development and support community is great; development tools become obsolete; source code is even lost!

The result is that we have programs of greater and greater size that are incomprehensible, understood neither by their authors nor by their maintainers.

In *Computer Power and Human Reason*, Joseph Weizenbaum (1976) asserts that this is a very dangerous phenomenon (p. 236):

> "Our society's growing reliance on computer systems that were initially intended to `help' people make analysis and decisions, but which have long since both surpassed the understanding of their users and become indispensable to them, is a very serious development. It has two important consequences. First, decisions are made with the aid of, and sometimes entirely by, computers whose programs no one any longer knows explicitly or understands. Hence no one can know the criteria or the rules on which such decisions are based. Second, the systems of rules and criteria that are embodied in such computer systems become immune to change, because, in the absence of a detailed understanding of the inner workings of a computer system, any substantial modification of it is very likely to render the whole system inoperative and possibly unrestorable. Such computer systems can therefore only grow. And their growth and the increasing reliance placed on them is then accompanied by an increasing legitimation of their `knowledge base.'"

Already our society's health is tightly coupled to computer programs that control vital functions such as the financial markets. For example, the design and linkage of computer-controlled financial systems has already contributed to wild fluctuations of the market (Sanger, 1987).

Software Engineering Approaches

The field of *software engineering* concerns itself with the technology and processes of software development, and thus it has approached the problems of software complexity and incomprehensibility in a number of ways.

The most widespread development has been the concern with the logical structure and expressive style of programs, resulting in modern software development techniques such as top-down design and stepwise refinement (Wirth, 1971), structured programming (Dahl, Dijkstra, and Hoare, 1972), modularity (Parnas, 1972), and software tools (Kernighan and Plauger, 1976).

A second advance has been the improvement in the clarity and expressive power of programming languages, as can be seen, for example, in Modula (Wirth, 1977) and Turing (Holt and Cordy, 1989), and in the development of object-oriented approaches to software design and development (Booch, 1991; Gamma, Helm, Johnson, and Vlissides, 1995).

There has also been progress in the organization and management of the team that produces the writing. This has given rise to concepts such as chief programmer teams (Baker, 1972), structured walkthroughs (Yourdon, 1979), and active design reviews (Parnas and Weiss, 1985).

The fourth development has been enhanced technology that supports the writing and maintaining of programs. This includes high-performance workstations and integrated software development environments (Wasserman, 1981; Dart, Ellison, Feiler, and Habermann, 1987).

Another important activity is CASE — computer-aided software engineering (Chikofsky and Rubenstein, 1988). Insights derived in the first four approaches are used to produce integrated environments in which programs can be created from specifications that are far terser and higher level than those required by conventional high-level languages.

A sixth more recent and related development is the attempt to build increasing amounts of knowledge and intelligence into software engineering tools and environments (Balzer, Cheatham, and Green, 1983; Barstow, 1987).

Enter Software Visualization

Yet despite these advances, the current appearance of programs typically:

• Does not contribute positively and significantly toward making a program easier to understand
• Does not reflect the history of a program as it has progressed through the software development cycle
• Does not facilitate the transfer of strategies and insights achieved by software developers to the ultimate readers and maintainers of the program
• Does not make important program structure as visible as it could
• Does not deal, therefore, with the fundamental problem of software comprehensibility, that of software complexity.

This motivates the seventh software engineering approach (Price, Baecker, and Small, 1993) — *software visualization*, which focuses on enhancing program *representation, presentation, and appearance*.

Visualization may be defined as “the power or process of forming a mental picture or vision of something not actually present to the sight” (Simpson and Weiner, 1989). Notice that this definition allows for the use of sensory modalities other than vision, e.g., hearing (see chapter by Brown and Hershberger), to assist in the formation of mental pictures or images.

Programmers have always employed pictures and diagrams informally as aids to conceiving, expressing, and communicating algorithms, as aids to illustrating *function*, *structure*, and *process*. If prepared thoughtfully, precisely, and imaginatively, typography, symbols, images, diagrams, and animation can present information more concisely and more effectively than the formal and natural languages typically used by the programmer.

In the remainder of this chapter, we shall sketch the early history of software visualization in terms of four major threads of activity:

• presentation of source code
• representations of data structures
• animation of program behaviour
• systems for software visualization.

A fifth important thread is the animation of concurrency (see chapter by Kraemer), but work in this area began relatively late.

Presentation of Source Code

An early attempt to improve program appearance was the development of a “presentation,” or “reference” form of the programming language ALGOL 60 (Naur, 1963). Another idea with a long history is *prettyprinting* (Baecker and Marcus, 1990, p. 18), the use of spacing, indentation, and layout to make source code easier to read in a structured language. *Prettyprinters* are programs that systematically indent the source code of a target program according to its syntactic structure. The earliest work was done on LISP, so that program readers would not drown in a sea of parentheses. Other early examples were NEATER2 (Conrow and Smith, 1970) for PL/I and Hueras and Ledgard's (1977) system for Pascal. The problems of prettyprinting Pascal elicited vigorous debate in early ACM SIGPLAN notices (Baecker and Marcus, 1990, p. 18).

More recent developments have used computerized typesetting and laser printing to improve the presentation of source code. The Vgrind utility of the Berkeley Unix system makes modest use of typographic encoding of keywords and user customizability of appearance. The Xerox Cedar user community has adopted a consistent publication style for softcopy and hardcopy listings of Cedar programs, making use of typeface, math notation, indentation, spatial separation, and headings (Teitelman, 1985; see also Baecker and Marcus, p. 20).

An ambitious recent attempt to enhance the presentation of source code is the work of Baecker and Marcus (1990, see chapter by Baecker and Marcus). Their

SEE Program Visualizer automatically typesets a C program according to an elaborate style guide based on graphic design principles. They also propose a method for documenting sets of C programs in a "program book." Knuth's (1984) WEB system also seeks to enhance program publishing, combining program source text and documentation in a single publication using a sophisticated markup language.

Diagramming Control Flow and Data Structures

The role of visual representations in understanding computer programs has a long history, beginning with Goldstein and von Neumann's (1947) demonstration of the usefulness of flowcharts. Haibt (1959) developed a system that could draw them automatically from Fortran or assembly language programs; Knuth (1963) produced a similar system which integrated documentation with the source code and could automatically generate flowcharts. Abrams (1968) is a review of such early systems. Although later experiments cast doubt on the value of flowcharts as an aid to comprehension (Shneiderman, 1980), recent results are more encouraging (Scanlan, 1989). The 1970's saw the first of many alternatives to flowcharting, the development of Nassi-Shneiderman diagrams (Nassi and Shneiderman, 1973) to counter the unstructured nature of standard flowcharts.

Baecker's (1968) prototype interactive debugger for the TX-2 computer produced static images of high-level language data structures and of the computer graphics display file. Articles by Stockham (1965) and by Evans and Darley (1966) review the then current state-of-the-art in debugging technology which motivated this work. Myers's (1983) Incense system was a more ambitious system for the display of data structures. Martin and McClure (1985) survey a variety of diagrammatic methods for the representation and display of program structure and behaviour.

More recently, there has been an explosion of interest in *visual programming*, the use of visual representations of programs as both an input and an output modality (Glinert, 1990a,b).

Animating Program Behaviour

Licklider did early experiments on the use of computer graphics to view how the contents of the memory of a computer were changing as the computer was executing. A different approach was taken with Knowlton's (1966a,b) influential films, which demonstrated L[6], Bell Lab's low-level list processing language. This work was the first to use animation techniques to portray program behaviour and the first to address the visualization of dynamically changing data structures.

Baecker, Hopgood, and Booth continued this work in pedagogical directions. Baecker (1973) outlined the potential of program animation and sketched many of the key research issues. Hopgood (1974) produced a series of short films illustrating hash coding and syntax analysis techniques. Yarwood (1974) explored the concept of program illustration, and methods of embedding graphical representations of program state within program source text. Booth (1975) produced a short film animating PQ-tree data structure algorithms. Baecker (1975) reported on work in which he and his students were investigating the portrayal of data structure abstractions and algorithms, eventually leading to the important film *Sorting Out Sorting* (Baecker, 1981; see chapter by Baecker).

Software Visualization Systems

The availability in the 1980's of personal workstations with bit-mapped displays and graphical user interfaces allowed researchers to go beyond the prototypes and specific animations of the 70s and develop software visualization systems. One of the earliest attempts to build a debugging system to aid visualization was the work done in Lisp by Lieberman (1984).

The most important and well known system of the new era was BALSA (Brown and Sedgewick, 1984), followed by Balsa-II (Brown, 1988a), which allowed students to interact with high level dynamic visualizations of Pascal programs. BALSA (see paper by Brown) evolved from a principled design, was used by hundreds of undergraduates and as a tool in algorithm design and analysis (Brown and Sedgewick, 1985; Brown, 1988b; see paper by Brown and Sedgewick), and was influential in inspiring many of the systems described in this volume.

Further Reading

Two good sources where one can continue reading about the history of software visualization research and development are Brown (1998a, Chapter 2), and Price, Baecker, and Small (1993).

References

Abrams, M.D. (1968). A Comparative Sampling of the Systems for Producing Computer-drawn Flowcharts, *Proceedings of the ACM National Conference*, 743-750.

Baecker, R.M. (1968). Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures, *Prof. First Hawaii International Conference on the System Sciences*, Jan, 1968, 128-129.

Baecker, R.M. (1973). Towards Animating Computer Programs: A First Progress Report, *Proceedings Third NRC Man-Computer Communications Conference*, May 30-31, 1973, 4.1-4.10.

Baecker, R.M. (1975). Two Systems which Produce Animated Representations of the Execution of Computer Programs, *SIGCSE Bulletin*, Vol. 7, No. 1, February, 1975, 158-167.

Baecker, R.M. (1981). With the assistance of Dave Sherman, *Sorting out Sorting*, 30 minute colour sound film, Dynamic Graphics Project, University of Toronto, 1981. (Excerpted and “reprinted” in *SIGGRAPH Video Review 7*, 1983.) (Distributed by Morgan Kaufmann, Publishers.)

Baecker, R.M., Grudin, J., Buxton, W., and Greenberg, S. (1995). *Readings in Human Computer Interaction: Toward the Year 2000.* Morgan Kaufmann.

Baecker, R.M. and Marcus, A. (1990). *Human Factors and Typography for More Readable Programs*. ACM Press, Addison-Wesley.

Baker, F.T. (1972). Chief Programmer Team Management of Production Programming, *IBM Systems Journal* 11(1), 56-73.

Balzer, R., Cheatham, T.E., and Green, C. (1983). Software Technology in the 1990's: Using a New Paradigm. *IEEE Computer* 16(11), 39-45.

Barstow, D. (1987). Artificial Intelligence and Software Engineering. *Proceedings 9th International Conference on Software Engineering*, 200-211.

Belady, L.A. and Lehman, M.M. (1976). A Model of Large Program Development. *IBM Systems Journal* 15(3).

Boehm, B.W. (1981). *Software Engineering Economics*. Prentice-Hall.

Booch, G. (1994). *Object Oriented Design with Applications*. Benjamin/Cummings.

Booth, K.S. (1975). *PQ Trees*, 12-minute colour silent film.

Brown, M.H. (1988a). *Algorithm Animation.* MIT Press.

Brown, M.H. (1988b). Exploring Algorithms Using Balsa-II. *IEEE Computer* 18(8), 27-35.

Brown, M.H. and Sedgewick, R. (1984). A System for Algorithm Animation. *Computer Graphics* 18(3), 177-186.

Brown, M.H. and Sedgewick, R. (1985). Techniques for Algorithm Animation. *IEEE Software* 2(1), January 1985, 28-39.

Chikofsky, E.J. and Rubenstein, B.L. (1988). CASE: Reliability Engineering for Information Systems. *IEEE Software* 5(2), 11-16.

Conroy, K. and Smith, R.G. (1970). NEATER2: A PL/I Source Statement Reformatter, *Communications of the ACM* 13, 669-675.

Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. (1972). *Structured Programming.* Academic Press.

Dart, S.A., Ellison, R.J., Feiler, P.H., and Habermann, A.N. (1987). Software Development Environments. *IEEE Computer* 20(11), 18-28.

Evans, T.G. and Darley, D.L. (1966). On-line Debugging Techniques: A Survey, *Proceedings of the Fall Joint Computer Conference* 29, 37-50.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Glinert, E. (Ed.) (1990a). *Visual Programming Environments: Applications and Issues.* IEEE Computer Society Press.

Glinert, E. (Ed.) (1990b). *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press.

Goldstein, H.H. and von Neumann, J. (1947). Planning and Coding Problems of an Electronic Computing Instrument. In *von Neumann, J., Collected Works* (A.H. Taub, Ed.), Macmillan, 80-151.

Haibt, L.M. (1959). A Program to Draw Multi-level Flowcharts. *Proceedings of the Western Joint Computer Conference*, San Francisco, 3-5 March, 131-137.

Holt, R.C. and Cordy, J.R. (1989). The Turing Programming Language. *Communications of the ACM* 31(12), 1410-1423.

Hopgood, F.R. (1974). Computer Animation Used as a Tool in Teaching Computer Science. *Proceedings IFIP Congress*, 889-892.

Hueras, J. and Ledgard, H. (1977). An Automatic Formatting Program for Pascal. *SIGPLAN Notices* 12(7), 82-84.

Kernighan, B.W. and Plauger, P.J. (1976). *Software Tools*. Addison-Wesley.

Knowlton, K. (1966a). *L6: Bell Telephone Laboratories Low-Level Linked List Language*. 16-minute black-and-white film, Murray Hill, N.J.

Knowlton, K. (1966b). *L6: Part II. An Example of L6 Programming*. 30-minute black-and-white film, Murray Hill, N.J.

Knuth, D.E. (1963). Computer-drawn Flowcharts. *Communications of the ACM* 6, 555-563.

Knuth, D.E. (1984). Literate Programming. *The Computer Journal* 27(2), 97-111.

Lieberman, H. (1984). Seeing What Your Programs Are Doing, *International Journal of Man-Machine Studies* 21(4), October 1984, 311-331.

Martin, J. and McClure, C. (1985). *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall.

Myers, B. (1983). Incense: A System for Displaying Data Structures. *Computer Graphics* 17(3), 115-125.

Nassi, I. and Shneiderman, B. (1973). Flowcharting Techniques for Structured Programming. *SIGPLAN Notices* 8(8), 12-26.

Naur, P. (Ed.) (1963). Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM* 6(1), 1-17.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* 15, 1053-1058.

Parnas, D.L. and Weiss, D.M. (1985). Active Design Reviews: Principles and Practices. *Proceedings of the 8th International Conference on Software Engineering*, August 1985, 132-136.

Price, B.A., Baecker, R.M., and Small, I.S. (1993). A Principled Taxonomy of Software Visualization, *Journal of Visual Languages and Computing* 4(3), September 1993, 211-266.

Sanger, D.E. (1987). The Computer Contribution to the Rise and Fall of Stocks, *New York Times*, December 15, 1.

Scanlan, D.A. (1989). Structured Flowcharts Outperform Pseudocode: An Experimental Comparison, *IEEE Software* 6(5), 28-36.

Shneiderman, B. (1980). *Software Psychology: Human Factors in Computer and Information Systems*. Little, Brown, and Co.

Simpson, J.A. and Weiner, C. (Eds.) (1989). *The Oxford English Dictionary*. Oxford University Press. XIX, 699-700.

Stockham, T.G., Jr. (1965). Some Methods of Graphical Debugging, *Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communications*, May 3-5, 57-71.

Teitelman, W. (1985). A Tour Through Cedar. *IEEE Transactions on Software Engineering* SE-11(3), March 1985, 285-302.

Wasserman, A.I. (1981). *Tutorial: Software Development Environments*. IEEE Computer Society Press.

Weizenbaum, J. (1986). *Computer Power and Human Reason*. W.H. Freeman.

Wirth, N. (1971). Program Development by Stepwise Refinement, *Communications of the ACM* 14(4), April 1971, 221-227.

Wirth, N. (1977). Modula: A Language for Modular Multiprogramming. *Software — Practice and Experience* 7(1), January 1977, 3-35.

Yarwood, Edward (1974). Toward Program Illustration, M.Sc. Thesis, Dept. of Computer Science, University of Toronto, Nov. 1974.

Yourdon, E. (1979). *Structured Walkthroughs*. Prentice-Hall.