# Software Visualization and Measurement in Software Engineering Education: An Experience Report

*James H. Cross II, T. Dean Hendrix, Karl S. Mathias, and Larry A. Barowski*
*Computer Science and Engineering*
*107 Dunstan Hall*
*Auburn University, AL 36849*
*grasp@eng.auburn.edu*

**Abstract –** *By supporting well-defined cognitive processes employed during a comprehension task, graphical representations of software could have a beneficial effect on comprehension efficiency and effectiveness. Documented empirical evidence of measurable benefits of software visualization is, however, limited in scope and often contradictory. The GRASP research project is currently evaluating the effects of software visualizations in both software engineering education as well as in software production environments. This paper describes the experimental framework and design used for evaluation in an academic setting.*

## Introduction

The idea that representing something visually can help our understanding has long been promoted in common practice and in the literature. Indeed, "a picture is worth a thousand words" has become a standard cliché. In the case of software, however, one must take great care that it is the correct thousand words that are being conveyed. Nonetheless, appropriate visualizations of software can be quite beneficial to programmers, especially when faced with program comprehension tasks. Such tasks exist throughout the software life cycle (e.g., formal technical reviews, debugging, verification, reverse engineering) and in the classroom (e.g., students reading examples from the text or examples from the professor).

The GRASP (Graphical Representations of Algorithms, Structures, and Processes) research project at Auburn University seeks to develop tools and techniques for the effective use of graphical representations and visualizations of software. The overall goal of this research is to increase the efficiency of programmer comprehension and understanding of source code. As an integral part of the research project, the GRASP software engineering tool has been developed as a continuously evolving prototype. The emphasis of the tool to this point has been on visualizing program structure and complexity via the automatic generation of software visualizations from source code. The current release of GRASP provides generation of these visualizations together with other program comprehension aids such as syntax coloring, typographical enhancements, and source code folding. When coupled with an appropriate compilation system, GRASP becomes an integrated graphical development environment, allowing users to edit, visualize, pretty-print, compile, link, execute and debug software.

Since GRASP was made available to the public, thousands of copies of the tool have been downloaded via anonymous file transfer protocol (FTP) and the World Wide Web (WWW) from educational, government, military and commercial sites, both in the United States and abroad. When it was released to the public, GRASP was also made available to users of the Auburn University College of Engineering computer network. GRASP is now used extensively throughout the computer science and engineering curriculum at Auburn University.

This paper describes the use of GRASP and software visualization in the context of software engineering education. It also describes an instrumentation framework that is being integrated into the local (i.e., Auburn) version of GRASP. This framework tracks individual utilization of GRASP and allows us to address questions such as "Under what situations do users perceive the CSD visualization to be useful in solving a problem?" and "Do software visualizations provide statistically significant gains in program comprehensibility?". Answers to these questions are an important problem in both educational research as well as software engineering research.

## Software Comprehension

Studies in software comprehension can be categorized into three main areas when it comes to comprehension: top down comprehension, bottom up comprehension, and a mixture of the two.

### Top Down

The idea that programmers use higher-level abstract constructs to assist in comprehending lower-level code originated with work conducted by Soloway, Bonar, and Ehrlich in studying how programmers comprehend looping constructs [12]. They found that programmers develop preferred strategies for certain types of loops. Preferred

**0-7803-5643-8/99/$10.00 © 1999 IEEE**

strategies required fewer cognitive elements to understand, and when programmers deviated from using these simpler strategies they made more errors.

Soloway and Ehrlich later extended the idea of looping strategies into a more general programming comprehension model of goals and plans [11]. A plan is a "canned" solution for a certain type of problem. As programmers develop skills, they develop a repertoire of plans that they are able to append, nest, and merge with each other to solve problems. Plans can be seen as the "mechanisms" put into the computer to solve the problem. Goals are then viewed as the "explanations" as to why the mechanisms exist.

#### Bottom Up

Pennington proposes that program knowledge is gained in a more bottom-up manner [8]. In this framework for comprehension, there are two models for representing knowledge: the program model and the situation model. She found that as people examine code, good comprehenders will switch between these models and poor comprehenders will stay with one or the other almost exclusively.

Pennington defines the program model as "a representation that highlights procedural program relations in the language of programs." The situation model is defined as "a representation that highlights functional relations between program parts that is expressed in the language of the domain world of objects." In other words, the program model explains how specific code constructs work, the situation model explains why the code is there to do that work.

#### Mixed Models

von Mayrhauser and Vans studied programmers working with large scale coding projects [13, 14]. They concluded that program comprehension involves the integration of four model components into a single framework. Their integrated code comprehension model includes a top-down domain model, a program model, a situation model, and a knowledge base. As the authors note, this is basically an integration of Soloway's top-down approach with Pennington's bottom-up approach.

## Improving Comprehensibility

Software visualization is an active area of research that investigates efficient and effective ways of automatically producing graphical representations of program source code, algorithms, or the runtime behavior of software. Such visualization technology promises to have a positive impact on software comprehension [2, 5]. By supporting well-defined cognitive processes employed during a comprehension task, graphical representations of software could have a beneficial effect on comprehension efficiency and effectiveness.

Visualization in and of itself, however, is not necessarily beneficial [9]. There are many issues that influence the utility of software visualization. Some are practical and cognitive issues relating to the user of the visualization and the process of human comprehension [1, 4, 13]. Other issues include those which relate to the nature of a particular visualization itself [6, 10].

The general goal of the GRASP research project is the investigation, formulation, generation, and evaluation of graphical representations of algorithms, structures, and processes which can be efficiently and effectively used to address real problems in software engineering in which program comprehension plays a central role. The GRASP prototype software engineering tool has been developed to automate the generation of the Control Structure Diagram (CSD), Complexity Profile Graph (CPG), and the Context View (CV) as fine-grained visualizations of source code.

The CSD is an algorithmic level graphical representation for software, the CPG is a new visualization of a fine-grained complexity metric, and the Context View is a visualization of an entire file condensed to fit in a small section of the screen. By synchronizing the CSD and the CPG, the CSD view of control structure, nesting and source code is directly linked to the corresponding visualization of statement level complexity in the CPG, while the CV constantly displays the current overall context of both the CSD and CPG. Currently, the CV is only available in a limited release research prototype.

#### Visualizing Control Structure

The CSD is the principal visualization in the GRASP environment. GRASP provides automatic generation of CSDs from Ada 95, C, C++, Java, and VHDL source code. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs should supplement the source code without disrupting its familiar appearance. That is, the CSD should appear to be a natural extension of the source code and, similarly, the source code should appear to be a natural extension of the diagram. This has resulted in a concise, compact graphical notation which attempts to combine the best features of diagramming with those of well-indented source code.

A comparison of the CSD with plain text source code is shown in Figure 1 and Figure 2. Figure 1 contains Ada 95 source code adapted from [3]. Figure 2 contains that same source code rendered as a CSD. While the same structural and control information is available in both figures, the CSD makes the control structures and control flow more visually apparent than does the plain text alone, and it does so

without disrupting the conventional layout of the source code.

The power of the CSD is much more evident in larger and/or more complex source code. For example in large programs, especially those which are a part of legacy systems, it is not uncommon for complex control structures such as loops to span hundreds of lines. The physical separation of sequential components within these large control structures becomes a significant obstacle to comprehension. The CSD clearly delineates each control structure and provides context and continuity for the sequential components nested inside, thus increasing comprehension efficiency. With additional levels of nesting and increased physical separation of sequential components, the visibility of control constructs and control paths becomes increasingly obscure, and the effort required of the reader can increase in the absence of the CSD.

```
task body TASK_NAME is
begin
   loop
      for p in PRIORITY loop
         select
            accept REQUEST(p) (D : DATA) do
               ACTION (D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end TASK_NAME;
```

*Figure 1. Ada 95 source code*

```
task body TASK_NAME is
begin
   loop
      for p in PRIORITY loop
         select
            accept REQUEST(p) (D : DATA) do
               ACTION (D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end TASK_NAME;
```

*Figure 2. Ada 95 source code rendered as a CSD*

**Visualizing Complexity and Context**

The Complexity Profile Graph is based on a set of functions that describes the context, the content and the scaling for complexity on a statement by statement basis [7]. When combined graphically, the result is a composite profile of complexity for the program unit. Ongoing research includes the development and refinement of the associated CPG functions and the development of the CPG generator prototype. A sample CPG for a program with approximately 3,700 lines of source code is shown in Figure 2. The spikes in the graph indicate areas of relatively high complexity.

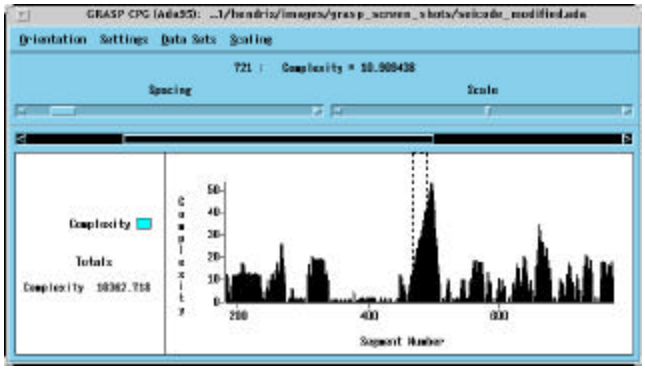


*Figure 3. Sample CPG*

The Context View (CV), the most recent GRASP visualization, was created for the purpose of providing a highly compressed view of an entire unit or file. Figure 3 shows a CSD window with a CV window docked on the right side. The CV window depicts a highly compressed view of the entire file, indicating the context of the full source code being displayed in the CSD window. Once fully developed, the CV will provide a user with a convenient and efficient method of navigating within the CSD and CPG windows.

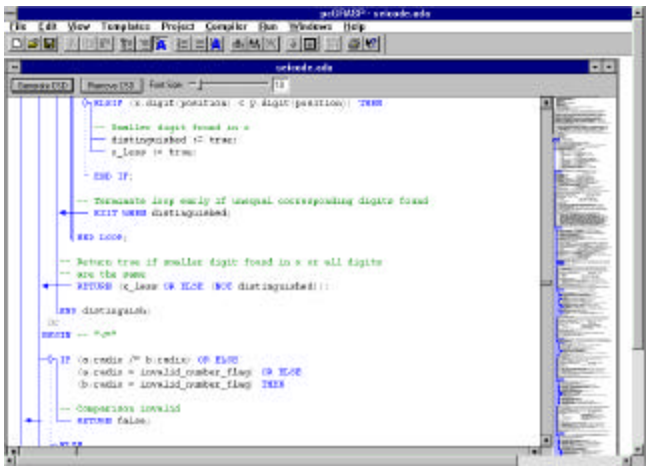

*Figure 4. CSD window with Context View*

## Initial Evaluation of Results

An important part of any research program is the evaluation of results. Fundamental evaluative questions that the GRASP research project must address include: Do users perceive a utility or benefit in using the software visualizations? To what extent and in what manner do users employ GRASP and its associated visualizations in real tasks? Do the visualizations provide statistically significant gains in program comprehensibility?

GRASP and its associated visualizations are used throughout the computer science and engineering (CSE) curriculum at Auburn University. Students in CSE courses ranging from beginning programming to graduate software engineering are encouraged and sometimes required to use GRASP in their lab and project work. This academic user population is being used for initial evaluations of GRASP. Later evaluations will be performed with an industrial user group.

### User Preference

Preference surveys have been used to informally assess the GRASP tool itself with respect to user satisfaction and degree of utilization. In an initial evaluation, a group of senior-level computer science students compared the CSD to four other graphical representations for algorithms (ANSI flowchart, Nassi-Shneiderman Diagram, Warnier-Orr Diagram, and Action Diagram) against eleven performance characteristics [6]. Statistical analysis of the data showed that the CSD was clearly preferred to all other graphical representations in a majority of the performance characteristics. Although this result is encouraging and is consistent with a large body of anecdotal evidence from GRASP users, user preference is not necessarily an objective measure.

### Utilization Analysis

To more objectively gauge how GRASP and its visualizations are being used, GRASP has been instrumented to automatically collect utilization data for every GRASP session that is run on the Auburn University College of Engineering network. Counter and timer variables associated with certain callback functions in GRASP have been made a part of the local (i.e., Auburn) GRASP executable. When these callbacks are made, the counters are incremented and the timers are started or stopped. When a user exits GRASP, the data contained in these counters and timers are sent via electronic mail to a special account where the data is stored in a text file. When a GRASP session is initiated on the College of Engineering network, both the GRASP executable and a daemon process are started. The daemon monitors the status of and communicates with GRASP to keep track of the counters and clocks. When a GRASP session ends, either via a normal exit or with a kill signal, received either explicitly or via a logout, the daemon sends the electronic mail that contains the utilization data and then kills itself. Local utilization data for GRASP has been collected in this manner since February 1996, and continues to the present. Thus, data exists for every session of GRASP run on the College of Engineering network for over three years.

An initial analysis of this utilization data has been performed. To investigate how GRASP usage is affected as students progress through the computer science and engineering curriculum, a subset of students were tracked from the introductory CS I course through subsequent junior level courses. The primary group under study consists of students enrolled in CSE200-220, Auburn's Ada-based CS1-CS2 sequence, during winter and spring quarters of 1997. Additionally, these students must have gone on to participate in at least one of two intermediate programming courses, CSE350, Systems Programming, or CSE360, Fundamental Algorithm Design and Analysis, during summer or fall quarters of that same year. The principle behind these decisions was to look for usage patterns as students were introduced to the tool and to the CSD during their initial programming course, and to see if these patterns carry over into their intermediate courses.

The initial analysis of this utilization is supportive of GRASP and the CSD. Although all students had ready access to alternative tools none of the students were required to use GRASP or its visualizations, all but two students showed heavy usage of GRASP in all four courses in the sequence. By default, the CSD is not generated for source code when a file is loaded in GRASP. A user must explicitly generate the CSD on demand, unless the default

setting is changed. Even so, student utilization of the CSD was also heavy. During the first two courses in the sequence, over three fourths of the students chose to view their source code with the CSD more than 50% of the time.

### Measuring Effects on Comprehension

*Anecdotal Evidence*

Besides user preference and tool utilization, learning outcomes are also being measured. Computer Programming for Engineers (CSE 120) is a required first programming course for all engineering students at Auburn University. CSE 120 introduces students to fundamental procedural programming skills using the C programming language. As an initial attempt to measure the effect of the CSD on student learning, two different sections of CSE 120 are being compared. Both sections are taught by the same instructor and the topical content, assignments, and exams are exactly the same. The only difference between the sections is the use of the CSD in instruction.

The first section was taught without the use of the CSD either by the instructor in class or by the students in lab. The second section was taught with the CSD being fully integrated into both the lecture and in the lab using GRASP. The same set of highly structured lecture notes, made available to the students, was used for both sections. For the CSD section, however, the source code in the notes was presented with the CSD added.

Learning outcomes for students were measured in terms of performance on graded items such as exams and programming assignments. Due to a lack of experimental controls a statistical analysis of the data was not performed. Information observation of the data, however, indicate potential benefits for students in the section that had the CSD integrated into lecture and lab.

*Repeatable, Controlled Experiments*

Software engineering experimental design is a difficult task and there are common errors that must be avoided. This research attempts to avoid such experimental design errors by (1) ensuring appropriate statistical controls, (2) conducting multiple runs of the experiments, (3) using appropriate sample sizes, and (4) utilizing both academic and industrial settings.

The planned experiments will be performed over a four year period and will be divided into four phases. In the first phase, which is almost complete, a pilot experiment was designed, where a single pair of comparable (programs) modules was utilized. That is, the two modules in the pair were similar in comprehensibility. For each subject, one module in the pair was selected randomly for presentation in plain text and the other in CSD form. Each subject evaluated both modules (CSD and plain text) in a completely randomized order. The subjects' evaluation of each module consisted of answering questions concerning structure and functionality of the source code. Student responses and timing data were automatically collected for each item as the experiment was administered to forty five CSE students. Statistical analysis of the data has only just begun, but an initial examination of the data looks promising.

This pilot experiment has one factor with two treatments (CSD and plain text) and another source of variation due to differences among (the) n subjects and hence involves a Repeated Measure Design (RMD). The statistical model for the pilot experiment is

$$y_{ij} = m + t_i + S_j + e_{ij}$$

where $y_{ij}$ represents the response from the j$^{th}$ subject (j = 1,2,…n) to the i$^{th}$ (i = 1,2) treatment, $m$ is the population mean, $t_i$ is the effect of the ith treatment, $S_j$ is the effect of the j$^{th}$ subject, and $e_{ij}$ represents experimental error in the (i,j) cell.

Results of previous experimental studies suggest that individual differences among subjects can account for a significant amount of variation in experimental performance. Other studies specifically attest to the importance of adequately controlling variation in knowledge of a particular tool or technique (such as GRASP and the CSD) when designing experiments with CASE tools. To address these threats to experimental validity, the GRASP environment has been instrumented to provide collection of general utilization data. This utilization data, along with other demographic data, was used to identify a suitably homogeneous pool of experimental participants from among the GRASP user population at Auburn University.

## Conclusion

The emphasis of the GRASP research project is on improving the comprehensibility of software through automatic generation of appropriate visualizations, such as the CSD, along with support for other comprehension aids such as folding. GRASP's unique combination of these features along with emerging features such as the context view, promises to be a highly effective aid to program comprehension. Indeed, if a graphical representation effectively supports well-defined cognitive processes employed during comprehension tasks, comprehension efficiency can be increased.

The current release of GRASP is being used extensively at Auburn University in a variety of computer science and engineering courses. Initial evaluations of the effectiveness of GRASP and its associated visualizations have been performed with highly encouraging results. Repeatable,

controlled experiments have begun to be performed with more in the planning stages. These experiments are currently being performed with CSE students, but ultimately will be refined and repeated with practicing software professionals as part of a research program funded by the National Science Foundation (EIA-9806777). The anticipated positive results of this research, coupled with detailed insights into how students and software professionals alike make use of software visualizations, could provide a catalyst for improvement of both pedagogy and industrial software practice.

## References


[1] Arunachalam, V. and Sasso, W. (1996). Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Engineering. *Journal of Systems and Software*, 34, pp. 177-189.

[2] Baecker, R. M., Digiano, C., and Marcus, A. (1997). Software Visualization for Debugging. *Communications Of The ACM*, 40, 4, pp. 44-54.

[3] Barnes, J. G. P. (1984) *Programming in Ada, Second Edition*, Menlo Park, CA: Addison-Wesley.

[4] Basili, V., and Selby, R. (1987). Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13, 12, pp.1278-1296.

[5] Cross, J. H. (1993). Improving Comprehensibility of Ada With Control Structure Diagrams. *Proceedings of Software Technology Conference*, April 11-14, 1994, Salt Lake City, UT (distributed on CD-ROM) 25 pages

[6] Cross, J.H., Maghsoodloo, S., and Hendrix, T.D. (1998). The Control Structure Diagram: An Initial Evaluation. *Empirical Software Engineering*, Vol. 3, No. 2, pp. 131-156.

[7] McQuaid, P. A., Chang, K. H. and Cross, J. H. (1995). Complexity Metric to Aid Software Testing and Maintenance. *Proceedings of Decision Sciences Institute*, 2, pp. 862-864.

[8] Pennington, N. (1987). Comprehension Strategies in Programming. in *Empirical Studies of Programmers: Second Workshop*, Ablex, pp. 100-112.

[9] Petre, Marian (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38, 6, pp. 33-44.

[10] Raymond, D. (1991). Characterizing Visual Languages. *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 176-182.

[11] Soloway, E. and Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10, 5, pp. 595-609.

[12] Soloway, E., J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," *Communications of the ACM*, 26(11), pp 853-860, November 1983.

[13] von Mayrhauser, A., and Vans, A.M. (1993). From Program Comprehension to Tool Requirements for an Industrial Environment. *Proceedings of the Second Workshop on Program Comprehension*, Capri, Italy, July, pp. 55-63.

[14] von Mayrhauser, A., and Vans, A.M. (1996). Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Transactions on Software Engineering*, 22, 6, pp. 424-437.