Do Visualizations Improve Program Comprehensibility? Experiments With Control Structure Diagrams for Java

T. Dean Hendrix, James H. Cross II, Saeed Maghsoodloo*, and Matthew L. McKinney
Computer Science and Software Engineering
(*Industrial and Systems Engineering)
Auburn University, AL 36849
{hendrix, cross, maghsood, mckinml}@eng.auburn.edu

Abstract

Recently, the first in a series of planned comprehension experiments was performed to measure the effect of the control structure diagram (CSD) on program comprehensibility. Upper-division computer science students were asked to respond to questions regarding the structure and execution of a source code module written in Java. Statistical analysis of the data collected from this experiment revealed that the CSD was highly significant in enhancing the subjects' performance in this program comprehension task. The results of this initial experiment along with the planned follow-on experiments promise to shed light on fundamental questions regarding the effect of software visualizations on program comprehensibility.

1 Introduction

Representing objects, processes, and ideas with pictures rather than words is intuitively appealing. The intuition is that a visual representation will be more readily understood than its textual counterpart. If one accepts such a premise, it is quite natural to investigate ways of applying visual representations to tasks in which comprehension plays a central role. Such tasks are abundant in the everyday world: For example, reading parts-assembly manuals to understand the structure of a machine, or reading operation manuals to understand how a machine works. In these particular domains, the utility of visual representations is accepted without question.

Applying visualization techniques to represent program structure and behavior is the central theme and focus of software visualization research. Although this area of research is quite active and graphical representations and visualizations for software abound, the effectiveness of software visualization is still an open question and is certainly not universally accepted.

SIGCSE 2000 3/00 Austin, TX, USA

Many experimental evaluations of software visualizations reported in the literature have indicated mixed results with perhaps a majority in the negative [2,3,4,5,6]. Although empirical studies of the usefulness of software visualizations generally show mixed results, other studies comparing the cognitive processing of simple pictures and text favor the efficiency of pictures. Numerous studies indicate that semantic analysis is performed faster for pictures than for text, and that graphical information is more easily and efficiently remembered than textual information [5,6]. These studies suggest that graphical representations of software are inherently useful, though particular representations may not be.

2 The Control Structure Diagram

The control structure diagram (CSD) is a graphical representation that visually depicts the control structure and module-level organization of source code [3]. 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 that attempts to combine the best features of diagramming with those of well-indented source code. Figure 1 illustrates the CSD with Java source code.

A comparison of the CSD with plain text source code is shown in Figure 2 and Figure 3. Figure 2 contains very "control-dense" Ada 95 source code adapted from [1]. Figure 3 contains that same source code rendered with 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 perhaps more evident in larger and/or more complex source code. For example in large programs, especially those which are a part of legacy systems or even those that upper-division computer science students create as part of their course work, it is not uncommon for complex control structures 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 potentially 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.

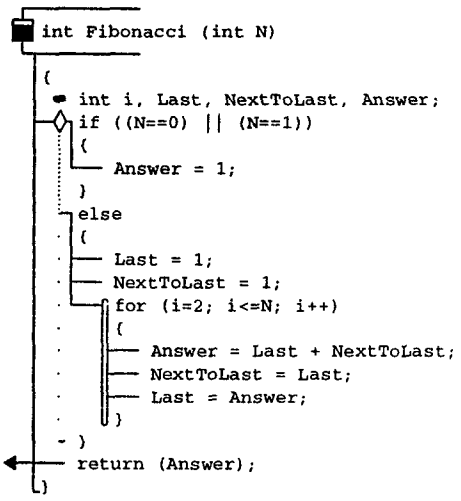

Figure 1. CSD with Java source code

```
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

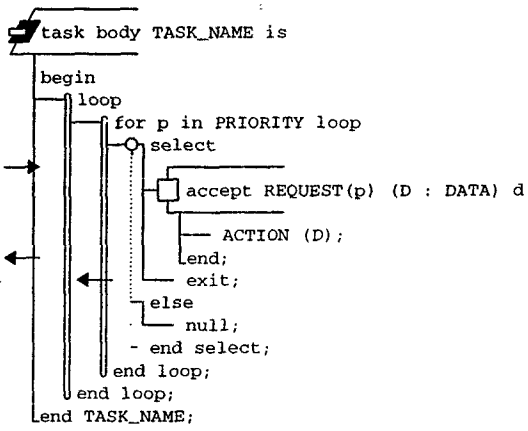

Figure 3. Ada 95 source code rendered as a CSD

It is clear from experience and from reports in the literature that a relationship exists between the syntactic form of source code and the ability of human readers to construct useful mental abstractions from that source code [2,6]. Source code that is well structured and visually appealing facilitates the comprehension process. The CSD, displayed as a companion to well-indented, pretty-printed source code should thus provide enhanced support for program comprehension.

3 Comprehension Experiment

Although the CSD was specifically designed to leverage the perceived advantages of a graphical representation together with the familiarity of pretty-printed source code, the success of this approach can only be determined by thorough, systematic evaluation procedures. Fundamental evaluative questions that must be addressed include: Do users perceive a utility or benefit in using the CSD? To what extent and in what manner do users employ the CSD in real tasks? Does the CSD provide statistically significant gains in program comprehensibility? These and other open questions are currently being addressed by the GRASP Research Project at Auburn University with funding from the National Science Foundation (EIA-9806777).

To measure the effect, if any, that the CSD has on program comprehensibility, a repeatable, controlled comprehension-based experiment was designed and implemented. Although a more detailed statistical analysis of the data remains to be completed, the initial results are quite promising and demonstrate that the CSD can provide statistically significant benefits in program comprehension tasks.

3.1 Procedure

In the experiment, subjects were presented with source code and asked questions relating to its structure and execution. The subjects were divided equally into two

groups. Both groups were presented with the same source code and asked to respond to the same series of 12 questions concerning the code. One group (the control) was given the source code in plain text only (as in Figure 2), while the other group was given the source code rendered with the CSD (as in Figure 3). Thus the independent variable is source code presentation (CSD or plain text). The task of each subject was to answer each question correctly in the shortest time possible.

The operational hypothesis is as follows:

H_1: The CSD will have a positive effect on program comprehensibility.

Thus, the null hypothesis that was tested is stated as:

H_0: The CSD will not have a positive effect on program comprehensibility.

Response time and response correctness are the two dependent variables. It is reasonable to assume that any effects of a visualization on comprehensibility would be manifested in at least one of these two measures. This assumption is also supported in the literature [4].

Both groups were given identical instructions concerning the completion of the experimental task prior to the beginning the experiment. In a 10 minute orientation session, subjects were provided with an overview of the task that they were being asked to perform. Each subject was presented with a short example program in laser-printed hardcopy form. They were then verbally provided with sample questions concerning the example program and informed of how they would be asked to record their response during the actual experiment. The group using the CSD had an additional 5-10 minute portion of the orientation session in which the basic symbols of the CSD were introduced and explained.

Both groups were told that the experimental task was to some extent designed to mimic elements of a software inspection or debugging activity, and thus were provided a motivational context for the experiment. Finally, each subject was given the fundamental instruction for the experiment: Without sacrificing accuracy, they were to answer each question as quickly as possible.

3.2 Participants

Students in an intensive upper-division object oriented programming course were asked to volunteer as subjects in the experiment. Volunteers were rewarded with extra credit points in the course. Using students from this course ensured that all the subjects were relatively expert at the experimental task: Each student had either senior standing or was a graduate student, and the course required each student to develop and debug non-trivial Java applications.

Since differences in ability among individual subjects in the groups could be a threat to experimental validity, the groups were balanced with respect to student performance in the course. At the time when the experiment was administered, the only graded item remaining in the course was the final exam. Thus, the performance balancing was done with almost complete grade information, thereby ensuring that the balancing was as accurate as possible. Figure 4 shows the performance balance between the two experimental groups. (A letter grade of 'X' indicates a graduate student taking the course on a non-letter grade basis.)

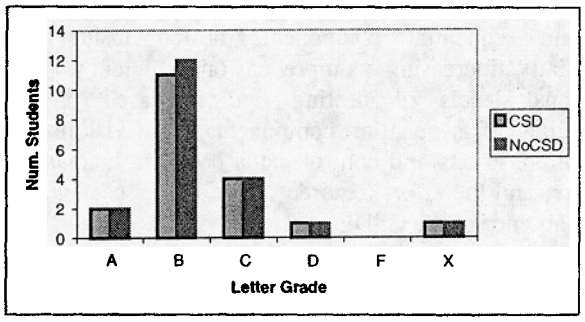

Figure 4. Performance Balance of Groups Prior to Experiment

Originally, 44 students volunteered to participate in the experiment. These 44 students were divided into two equal sized groups and performance balanced as discussed above. When the experiment was administered, however, some of the volunteers were absent. This made the two groups unbalanced both in number and in performance. Specifically, the CSD group had twice as many A students as the control group (4 versus 2) and the control group had an F student where the CSD group had none. To bring the two groups back into balance, data from the two A students in the CSD group who had the best performances in the experiment and the F student from the control group were eliminated before the data were analyzed. Thus, data from only 39 subjects (with group balancing as shown in Figure 4) were made available for analysis.

3.3 Questions and Presentation

In the interest of making the experimental task as realistic and practical as possible, a module from a public domain graphics package was selected to be the source code under inspection. The package was written in Java and the selected module was a function containing 183 source lines of code with several levels of control. The function had a small number of control constructs added for the purposes of the experiment, but was otherwise unchanged. Both groups were made aware during the orientation session that they were inspecting "real" code and not something that had been manufactured for the experiment.

To eliminate the effect that individual familiarity with a particular program editor might have on the experimental results, both groups were given the source code in laser-printed hardcopy form.

To facilitate accurate and efficient recording of responses and response times, the questions were presented to the subjects in a sequence of web pages. Each web page contained a single question along with a text field and a submit button. To respond to a question, a subject simply typed in their answer in the text field and clicked on the submit button. A script associated with each web page automatically recorded the subject's response as well as the response time for that question. The response time was calculated as the amount of elapsed time from when the question was displayed to when the subject submitted a response.

The questions were designed according to several criteria: (1) The questions should be relevant to completing real comprehension tasks such as those found in inspection, testing, maintenance, and debugging activities. The experimental questions should be similar if not identical to real, practical questions concerning the source code. (2) The questions should be universal, or as generally applicable as possible. That is, the questions should be drawn from a set that would have to be answered, either explicitly or implicitly, in most program comprehension tasks regardless of the task context or program functionality. For example, questions concerning the syntactical boundaries of constructs and questions concerning transfer of control after a certain point in execution fall into this category. (3) The questions should have single, objective answers. The questions were designed to be answered in terms of line numbers in the source code, and are thus unambiguous and easily scored.

Representative questions from the experiment include:

1. Where does the loop that begins on line 91 end?
2. How many variables and object instances are declared?
3. How many ways are there to exit the loop that begins on line 91?
4. To what line would control be transferred immediately after executing line 144?
5. How many syntactic levels deep is the most deeply nested statement?
6. How many conditions must be evaluated in order for line 152 to be executed?

3.4 Results

Analysis of the data strongly rejected the null hypothesis that the CSD had no positive effect on subject performance in answering the 12 questions. Indeed, the effect of the CSD on both the speed and correctness of responses was significant.

An initial analysis of differences in performance between the two groups was done using average time taken to respond to each question (T1), average time taken to respond correctly to each question (T2), and number of correct responses across all questions (T3).

Figure 5 graphs the average response time without regard to correctness (T1). There is only one question (number 12) for which the control group performs better. But this must be understood in light of the fact that there were no correct responses from the plain text control group for question 12. The positive effect of the CSD on overall response time (T1) is significant at the 0.06 level.

Figure 6 graphs the average response time for correct responses (T2). Here, the control group never outperforms the CSD group and the positive effect of the CSD is highly significant at the 0.0013 level. It should be noted that this result is strengthened by the fact that for three questions (6, 7, and 12) there were no correct responses from the control group. The graph in Figure 6 selects the average response time for the control group on those questions.

Figure 7 graphs the total number of correct responses per question for each group (T3). Again, the performance gain of the CSD group is significant: 45% of the CSD group's responses were correct while only 26% of the control group's responses were correct.

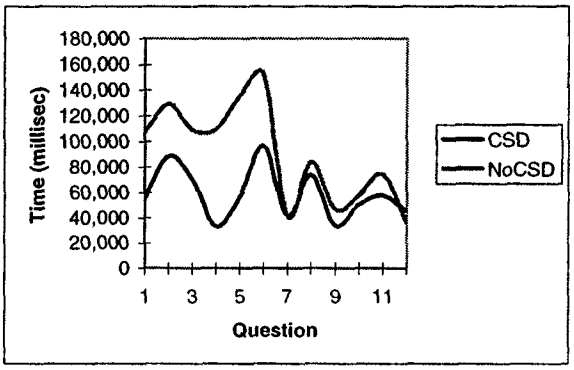

Figure 5. Time Taken to Respond

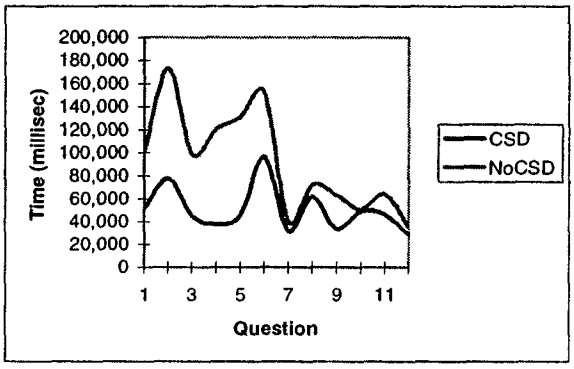

Figure 6. Time Taken to Respond Correctly

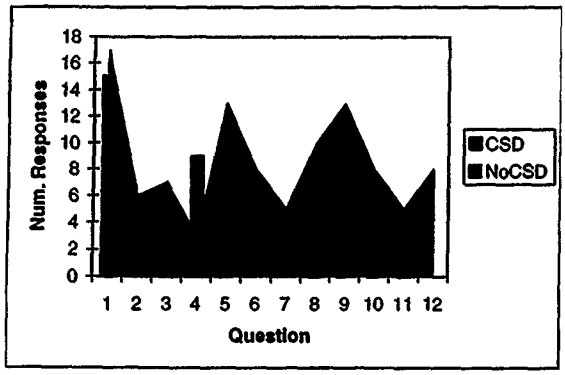

Figure 7. Number of Correct Responses

A rigorous statistical analysis of the data has just begun and will be reported in a subsequent publication. These initial results, however, are quite promising. Follow-on experiments are planned that will build on these results and will further explore the human performance benfits offered by certain software visualizations.

4 GRASP

Benefits notwithstanding, unless a visualization can be efficiently rendered in a program editing environment, it is highly unlikely that the visualization will be used in practice. GRASP (Graphical Representation of Algorithms, Structures, and Processes) is a software engineering tool that automatically generates the CSD for multiple languages with sufficient speed for use even in production environments. Currently, GRASP processes Java, C++, C, Ada, and VHDL, producing the CSD for each language at approximately 20,000 lines of code per second.

GRASP executables are freely available for multiple platforms (currently Windows 95/98/NT, Linux, Solaris, SunOS, IRIX, AIX, and various other flavors of UNIX) at the URL http://www.eng.auburn.edu/grasp. There is also a pure Java implementation of GRASP that runs on the JVM (see Figure 8). Each binary distribution of GRASP offers automatic generation of the CSD for all the supported languages as well as automatic generation of other visualizations, such as the complexity profile graph, in some cases. In addition to generating the visualizations, GRASP also offers a fully-functional program text editor with syntax coloring, syntax templates, and construct folding. When combined with an appropriate compilation system (such as gcc for C/C++/Ada, or Sun's JDK), GRASP becomes a complete program development environment with support for program editing, visualization, compilation, and runtime execution monitoring.

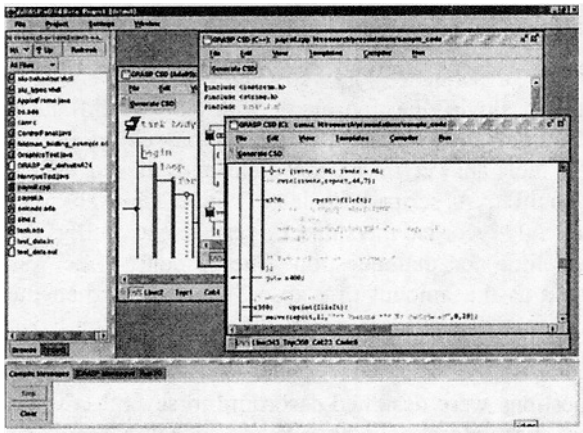

Figure 8. GRASP for the JVM

5 Summary

Effective software visualizations can provide measurable benefits in program comprehension tasks. Since such tasks occur both in practice and in the classroom, tools that efficiently generate such visualizations can be valuable aids to both software professionals and students alike.

The CSD is a graphical representation designed to be a companion to rather than a replacement for source code. The results of a controlled experiment indicate that the CSD can have a highly significant positive effect on human performance in program comprehension tasks. These results together with the results from planned follow-on experiments promise to address important open questions in both software visualization and software engineering research.

References

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

[2] Cant, S.N., Jeffery, D.R., and Henderson-Sellers, B. (1995). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, 37 (7), pp. 351-362.

[3] 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.

[4] Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J, and Boehm-Davis, D.A. (1989). Experimental Evaluation of Software Documentation Formats. *The Journal of Systems and Software*, Vol. 9, pp. 167-207.

[5] Goolkasian, Paula (1996). Picture-Word Differences in a Sentence Verification Task. *Memory & Cognition*, 24, 584-594.

[6] Hendrix, T.D., Cross, J.H., Barowski, L.A., and Mathias, K.S. (1998). Visual Support for Incremental Abstraction and Refinement in Ada 95. *Proceedings of SIGAda '98*, Washington, D.C., November 10-12, 1998.