The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities

Dean Hendrix, *Member*, *IEEE*, James H. Cross II, *Senior Member*, *IEEE*, and Saeed Maghsoodloo

Abstract—Recently, the first two in a series of planned comprehension experiments were performed to measure the effect of the control structure diagram (CSD) on program comprehensibility. Upper- and lower-division computer science and software engineering students were asked to respond to questions regarding the structure and execution of one source code module of a public domain graphics library. The time taken for each response and the correctness of each response was recorded. Statistical analysis of the data collected from these two experiments revealed that the CSD was highly significant in enhancing the subjects' performance in this program comprehension task. The results of these initial experiments promise to shed light on fundamental questions regarding the effect of software visualizations on program comprehensibility.

Index Terms—Software visualization, control structure diagram, program comprehension, controlled experiments, evaluation.

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: e.g., 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. The roles of visualization in software design, implementation, formal technical reviews, and maintenance are of particular interest. Specifically, those activities in which program comprehension plays a significant role are expected to benefit through the effective use of software visualizations. Since it has been estimated that in the maintenance phase alone software professionals spend at least half of their time analyzing software artifacts in an attempt to comprehend the software [13] and that code reading is a popular and viable verification and testing strategy [5], [8], it follows that increasing the comprehensibility of software should have a significant impact on improving productivity and reducing cost. 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. Von Mayrhauser and Vans [40], [41] studied the cognitive limitations that hindered software engineers in large maintenance projects. Their study indicated both the need for automated comprehension support tools in which software visualizations could play an important role and also the inadequacy of current tools.

There are many issues that influence the utility of software visualization. Some issues are practical and cognitive relating to the user of the visualization and the process of human comprehension [2], [31], [39], [41]. Other issues include those which relate to the nature of a particular visualization itself [34].

This paper describes experiments performed to measure the effect of a particular software visualization on performance in source code comprehension activities, and in so doing makes the following contributions:

- The experimental design focuses on comprehension activities that are common to most, if not all, source code comprehension tasks and most cognitive models of program comprehension. Thus, the experiments are applicable to a variety of tasks (e.g., source code inspections, source level debugging) and to a variety of comprehension strategies (e.g., top-down, bottom-up).
- The experimental design focuses on fine-grained comprehension activities that are considered low-level or primitive components of higher-level comprehension strategies. Thus, instead of revealing a positive effect while shedding little or no light on why the results were obtained, these experiments

- *D. Hendrix and J.H. Cross II are with the Computer Science and Software Engineering Department, Auburn University, AL 36849. E-mail: {hendrix, cross}@eng.auburn.edu.*
- *S. Maghsoodloo is with the Industrial and Systems Engineering Department, Auburn University, AL 36849. E-mail: maghsood@eng.auburn.edu.*

Manuscript received 20 June 2000; revised 9 Apr. 2001; accepted 19 Apr. 2001.
Recommended for acceptance by P. Johnson.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112330.

allow results to be directly associated with specific, well-defined comprehension activities.
- The experiments lay the groundwork for a larger research program that incrementally scales from fine-grained comprehension activities to more coarse grained, complex tasks. Such a systematic research program is needed to clarify the present understanding of the effectiveness of visualization techniques for software.

The remainder of the paper is structured as follows: Section 2 discusses related research, points out the need for experimentation in this area, and describes the particular focus and approach taken in the present work. Section 3 describes the control structure diagram, which is the particular software visualization under study. Section 4 contains a description of the experiments performed along with detailed analyses of the data. Section 5 draws conclusions and outlines future work.

2 Motivation for the Research

While the potential utility of visualizations may appear to be widely accepted, the extent of their use in practice is not clear. Documented empirical evidence of measurable benefits of software visualization is limited in scope and contradictory, especially with respect to production software [31].

2.1 Need for Experimentation

The fundamental issue being addressed—the relationship between software visualizations and the human comprehension of software—is intrinsically one that requires experimentation. The literature on software visualizations is filled with graphical representations of control structure, data structure, architecture, and class and object hierarchy [32], [33]. These visualizations range from strictly static views of source code to dynamic views of the runtime behavior of an algorithm. The proponents of these visualizations and their associated automated environments generally make claims for potential improvements to various software development activities afforded by the visualizations. While visualizations abound, empirical evaluations of their effectiveness have been generally limited in scope and not necessarily designed to scale up to industrial practice.

Numerous evaluations reported in the literature have indicated mixed results with many in the negative. For example, Aoyama [1] and Scanlan [36] indicated gains in comprehensibility when using visualizations while Green et al. [21] and Green and Petre [20] reported that graphics were significantly slower than text in the experimental comprehension tasks. Scanlan's experiments compared plain text algorithms with structured flowcharts. Green's experiments included the comparison of indented plain text algorithms with a gate type diagram (digital design) and an interactive sequential block diagram. Moher et al. [25] used three forms of a Petri net in a similar study and reported that none of the graphical representations were significantly better than the corresponding textual version. Petre [31] asserted that secondary notation, such as spacing and shape of symbols, plays a significant role in the effect of a graphical representation. Furthermore, Petre claimed that comprehensibility is affected mainly by this secondary notation rather than the primary graphical constructs and symbols that comprise the representation. These studies also showed that poor use of secondary notation can confuse and mislead one when reading the graphical representation.

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 a pictorial representation. 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 [19], [43]. These studies suggest that graphical representations of software are inherently useful, though particular representations may not be. For example, studies reported by Green and Petre [20] and Moher et al. [25] each showed that properly indented plain text outperformed or performed as well as the graphical representations they tested. An examination of the particular visual representations used in their experiments, however, indicated that they were far from intuitive or familiar.

Experimentation is also needed to address the interaction of visualizations with various cognitive models of program comprehension [43], [27], [12]. Pennington [30] asserts that programmers attempt to comprehend source code in a bottom-up fashion. That is, small sections of code are understood, followed by large blocks based on control flow relationships. Other studies [38], however, assert that, in some situations, programmers employ a top-down comprehension strategy. Such a strategy would begin at a high level of abstraction (e.g., architecture) and work toward comprehending lower levels of abstraction. Von Mayrhauser and Vans [41] have defined an integrated comprehension model that combines the top-down and bottom-up strategies. Their work indicates the importance of tools to support comprehension tasks at both high levels of abstraction (e.g., architecture) and lower levels of abstraction (e.g., source code). Citrin et al. [12], however, report the difficulty in providing visualizations at both high and low levels of abstraction and point out the need for scalable visualizations.

Murphy et al. [27] reported a study of several broadly distributed software engineering tools which generate call graphs from existing source code as an architectural level visualization. The call graphs generated by the different tools varied greatly, even when applied to the same source code. The authors of the study observed that, although the architectural level visualization tools obviously made different design decisions when planning their visualizations, there were insufficient data on how software professionals actually use and benefit from such visualizations to make a determination of good or poor choices. Studies are needed to identify software engineering tasks that could benefit from particular types of visualizations as well as appropriate information that is needed to support the automatic generation of such visualizations. This knowledge would allow appropriate visualization tools to be developed [27].

2.2 Focus and Approach

The role played by visualizations in software comprehension tasks is certainly not well understood. The work reported in this paper seeks to clarify current understanding on this issue by performing controlled experiments that can yield important, baseline information and also be used as springboards to future, more comprehensive studies. The approach taken is to focus the initial experiments on the effectiveness of a particular visualization with respect to fundamental comprehension activities that are applicable to a wide variety of comprehension tasks as well as components of a variety of comprehension models.

One commonly accepted element of program comprehension can be referred to as incremental abstraction, that is, the process of successively building up mental representations of various levels of abstraction of source code text structures and their relationships. These mental representations are often called *chunks* and have been shown in the literature to play significant roles in human comprehension of software [11], [18], [29].

The study of fundamental cognitive processes has revealed well-defined limits on the capacity of the human mind during comprehension tasks. Miller [24] described the classic 7 ± 2 limit on short-term memory and others have demonstrated that this capacity diminishes as task complexity increases [22]. Thus, grouping portions of source code together allows a section of code to be understood abstractly as a unit rather than through the complete details of its components. This concept builds upon itself in that lower-level groupings are combined to form higher-level units, thus providing a method of comprehending a program in terms of various levels of abstraction in a systematic way, while addressing the inherent limits on the capacity of the human mind.

Before chunks can be used as part of a comprehension strategy, they must be identified. This process of identification, which involves scanning through the source code in either a forward or backward direction, is known as *tracing* [11]. Tracing involves both semantic and syntactic knowledge [37]. Semantic knowledge is relatively independent of any particular language and involves understanding basic concepts such as looping structures and fundamental algorithms and recognizing design patterns in code. Syntactic knowledge is language specific and allows semantic structures to be recognized in a particular language [11]. Since the process of tracing is fundamental to so many comprehension tasks in a variety of contexts, it was selected as the comprehension activity on which to focus the initial experiments.

The control structure diagram (CSD) was selected as the particular visualization under study. The CSD is a graphical representation that visually depicts the control structure and module-level organization of a program. The CSD was designed to address some of the possible shortcomings of other algorithmic-level visualizations such as flowcharts. Visualizations such as flowcharts disrupt the layout of the source code by viewing a program as a flow graph with source code statements or statement fragments attached to nodes of the graph. Thus a reader is forced to comprehend the source code through a completely different notation and layout (i.e., the flowchart). The CSD appears as a companion to rather than a replacement for source code, thus leveraging the perceived advantages of a graphical representation together with the familiarity of pretty-printed source code. In an earlier study [15], the CSD was compared to four other graphical representations which were considered to be representative of a group of well-known algorithmic level visualizations at the time: ANSI flowchart, Nassi-Shneiderman Diagram, Warnier-Orr Diagram, and Action Diagram. The study focused on detailed comparisons of the notations against 11 performance characteristics and a preference-based instrument was given to 33 upper-level software engineering students at Auburn University. The CSD was clearly preferred to all other graphical representations in a majority of the performance characteristics. There is also significant anecdotal evidence from CSD users in industry that the diagram provides a definite benefit in code reading and comprehension tasks.

Thus, the focus of the experiments reported in this paper is on measuring the effect, if any, of the CSD on program comprehensibility, specifically with respect to tracing. This approach allows the experiments to address broadly applicable comprehension activities while providing a natural starting point for a larger, more comprehensive research program.

3 The Control Structure Diagram

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. Indeed, since many professional programmers consider the source code to be the only trusted specification of the software [42], the CSD was designed to seamlessly coexist with the source code. This has resulted in a concise and compact graphical notation that attempts to combine the best features of diagramming with those of well-indented source code (see Fig. 1).

For illustrative purposes, a comparison of the CSD with plain text source code is shown in Figs. 1 and 2. Fig. 1 contains Java source code and Fig. 2 contains the 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.

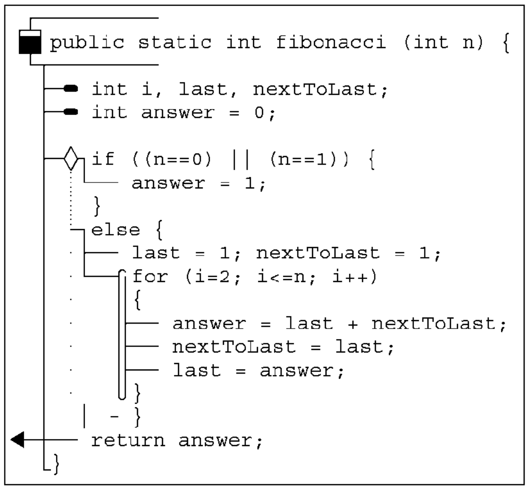
The CSD provides companion graphical representations for all the major control constructs found in Java, Ada 95, C, and C++. Appendix A illustrates the CSD for Java's major control structures, while a more complete discussion of the CSD is given in [14].

The CSD's utility is perhaps more evident in larger and/or more complex programs. For example, in large programs, especially those that are a part of legacy systems, 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

```
public static int fibonacci (int n) {
   int i, last, nextToLast;
   int answer = 0;

   if ((n==0) || (n==1)) {
      answer = 1;
   }
   else {
      last = 1; nextToLast = 1;
      for (i=2; i<=n; i++)
      {
         answer = last + nextToLast;
         nextToLast = last;
         last = answer;
      }
   }
   return answer;
}
```

Fig. 1. Plain text source code.



```
public static int fibonacci (int n) {
   int i, last, nextToLast;
   int answer = 0;

   if ((n==0) || (n==1)) {
      answer = 1;
   }
   else {
      last = 1; nextToLast = 1;
      for (i=2; i<=n; i++)
      {
         answer = last + nextToLast;
         nextToLast = last;
         last = answer;
      }
   - }
   return answer;
}
```

Fig. 2. Source code rendered with a CSD.

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.

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 programmers to construct useful mental abstractions from that source code [11], [14]. Source code that is well structured and visually appealing facilitates the comprehension process. The use of good typography in program comprehension has been documented extensively in the literature [10], [5], [23], [28]. Bouwhuis [10] found that good comprehension is the result of deliberate reading, which takes advantage of good typography and other semantic cues. Deliberate reading should be facilitated by the graphical constructs that are in addition to existing typographic conventions and indentation in source code. Since the CSD adds visual cues for sequence, selection, iteration, exits, exceptions, etc., it should improve the clarity of plain text.

The validity of these claims concerning the CSD 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 CSD utilization provide statistically significant gains in program comprehensibility?

4 Comprehension Experiment

To measure the effect, if any, that the CSD has on program comprehensibility, two controlled comprehension-based experiments, I and II, were designed and implemented. Experiment I involved a senior-level software engineering class that also had a few graduate students and Experiment II dealt with a sophomore-level class that involved only undergraduates with little experience in programming. The results from these two experiments are quite promising and demonstrate that the CSD can provide statistically significant benefits in program comprehension tasks.

4.1 Procedure

The students of Experiment I in the senior-level course and those of Experiment II in the sophomore-level course were presented with the same Java source code and asked questions related to its structure and execution. The subjects were divided equally into two subgroups. Both subgroups were presented with the same source code and asked to respond to the same series of 12 questions concerning the code. One subgroup (the control) was given the source code in plain text only (as in Fig. 1, i.e., without the CSD), while the experimental subgroup was given the source code rendered with the CSD (as in Fig. 2). Thus, one factor was source code presentation (with CSD = level 1, or in plain text = level 0). The task of each subject was to answer each question correctly in the shortest time possible.

The operational (or alternative) hypothesis is as follows:

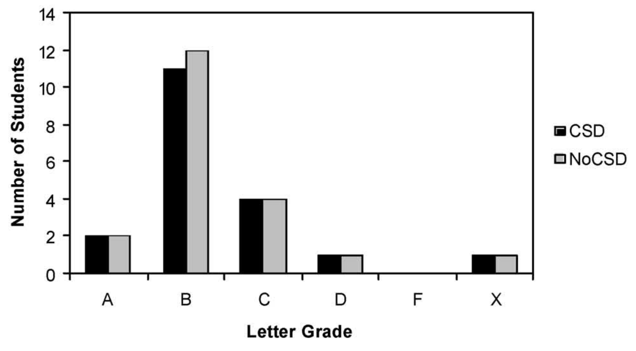
H_1: The CSD did have a positive effect on program comprehensibility.

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

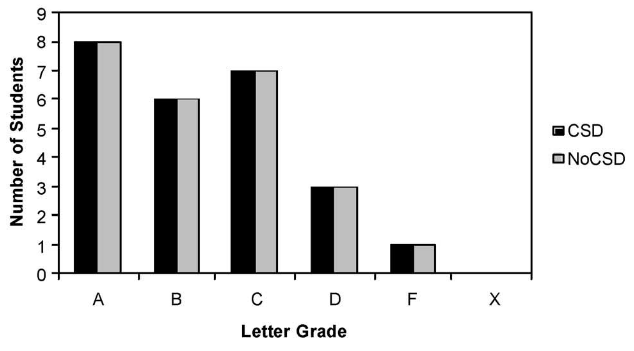
H_0: The CSD did not have a positive effect on program comprehensibility.

Response time and response correctness were 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 [17]. Response correctness was measured by whether the answer to a question was correct or not.

Both subgroups in the two experiments were given identical instructions concerning completion of the experimental task prior to beginning the experiments. 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



(a)



(b)

Fig. 3. Performance balance of groups prior to the experiments. (a) Experiment I and (b) Experiment II.

asked to record their response during the actual experiment. The subgroup using the CSD had an additional 5 to 10 minute portion of the orientation session in which the basic symbols of the CSD were introduced and explained.

The groups in both experiments 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.

4.2 Participants

In both experiments, students from two different courses, both of which required an extensive amount of programming in Java, were asked to volunteer as subjects in the experiments. Volunteers were rewarded with extra credit points for their participation. Using students from the same course in each of the two experiments ensured that all subjects were either relatively expert at the experimental task (all students in the senior-level class of Experiment I were either a senior or a graduate student), or all were inexperienced in programming (all subjects in the sophomore-level class were Freshmen or sophomores).

Since differences in ability among individual subjects in the two groups could be a threat to experimental validity, the two subgroups in both courses were balanced randomly with respect to student performance in the course. At the time when the experiments were administered, the only graded item remaining in both courses were the final exams. Thus, the performance balancing was done with almost complete grade information, thereby ensuring that the balancing was as accurate as possible. Since both courses were programming intensive and based on Java, student performance in these courses was directly related to the experimental task. Figs. 3a and 3b show the performance balance between the two subgroups for the Experiment I and II, respectively. (A letter grade of "X" indicates a graduate student taking the course on a nonletter grade basis.)

Originally, 44 students in the senior-level and 50 students in the sophomore-level course volunteered to participate in the two experiments. These students were divided into two equal sized subgroups and performance balanced as discussed above. When the experiment was administered, however, some of the volunteers in the senior-level class were absent. This made the two subgroups of that class unbalanced both in number and in performance. Specifically, the CSD subgroup had twice as many A students as the control subgroup (4 versus 2) and the control subgroup had an F student while the CSD subgroup had none. It was felt that this imbalance, particularly with respect to student performance, could pose a threat to the validity of the experimental results. To bring the two subgroups back into balance and address this possible threat to validity, the data from the two A students in the CSD subgroup who had the best performances in the experiment and the F student from the control subgroup were eliminated before the data were analyzed. Thus, data from only 39 subjects (with subgroup balancing as shown in Fig. 3) were made available for analysis. Further, one student in the control subgroup (i.e., without the CSD) of the senior-class made a procedural error in reporting and, therefore, his/her data point was completely disregarded in the analysis. Therefore, the senior-level group of Experiment I had 38 participants divided into two subgroups (control and experimental) of 19 students each. No procedural problems were encountered with the sophomore-level group so that both the control and experimental subgroups of Experiment II consisted of 25 students each.

4.3 Questions and Presentation

For the experimental task to be as realistic and practical as possible, the source code under inspection was selected from a graphics package currently in use as part of a data analysis and presentation tool. The package, which was written in Java, contained a function that with only minor modifications could be made suitable for stand-alone presentation in the experiments. This function contained 183 lines of source code[1] and exhibited several types of control structures. To eliminate the effect that individual familiarity with a particular program editor might have on

1. Due to its length, the actual source code used in the experiemnt is not included in this paper. The complete source code is available on request through email to D. Hendrix.

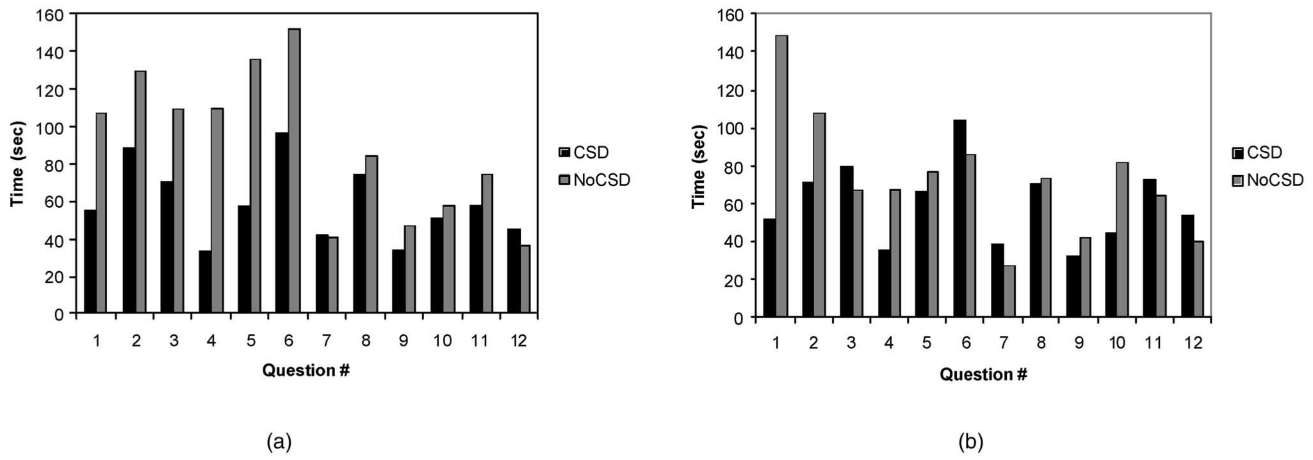


Fig. 4. Total Average Time Taken to Respond. (a) Experiment I: The difference $\bar{\mathbf{T}}_1$ (NoCSD) - $\bar{\mathbf{T}}_1$ (CSD) = 31.54 was significant at the 0.0216 level and (b) Experiment II: The difference $\bar{\mathbf{T}}_1$ (NoCSD) - $\bar{\mathbf{T}}_1$ (CSD) = 13.50 was not significant at the 0.1876 level.

the experimental results, both groups in the experiments were given the source code in laser-printed hardcopy form.

To facilitate accurate and efficient recording of response times and whether the corresponding answer was correct or not, 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 variable, **T**, was calculated as the amount of elapsed time from the instant a question was displayed to when the subject submitted the corresponding response. To eliminate any effect that the order in which the subjects responded to the questions might have on their performance, the questions were presented in a random order. The same random order was used for all subjects.

The questions were designed in accordance with the focus and approach to the experiment described in Section 2.2. The questions should focus on the process of tracing while being relevant to real comprehension tasks such as those found in inspection, testing, maintenance, and debugging activities. 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. In addition, the questions were also designed to be answered in terms of line numbers in the source code and were thus unambiguous and easily scored.

Experimental questions were thus written to address the following categories of program knowledge, related directly to tracing:

- Syntactical boundaries of control constructs (e.g., where does a given loop end?).
- Location and number of entrance and exit points for control constructs (e.g., how many exit points does a given loop have?).
- The target for a transfer of control after statement execution (e.g., where does execution continue after a given statement?).
- Syntactical nesting depth (e.g., how many levels deep is a given statement?).
- Execution predicates for statements (e.g., how many conditions must be evaluated for a given statement to execute?).

4.4 Results

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

An initial analysis of differences in performance between the two subgroups was done using average time taken to respond to each question ($\bar{\mathbf{T}}_1$), average time taken to respond correctly to each question ($\bar{\mathbf{T}}_2$), and the number of correct responses across all questions (**X**).

4.4.1 Analysis of Average Response Times ($\bar{\mathbf{T}}_1$)

Figs. 4a and 4b graph the total average response time ($\bar{\mathbf{T}}_1$) over all 12 questions, without regard to correctness. In Experiment I (Fig. 4a), there is only one question (number 12) for which the control group performed better, while the control group performed a bit better in questions 3, 6, 7, 11, and 12 (i.e., five out of 12 questions) in Experiment II (Fig. 4b). This must be understood, however, in light of the fact that there were no correct responses from the plain text control group for question 12 in Experiment I, while, in Experiment II, the CSD subgroup had 44 correct answers versus only seven correct answers for the control (NOCSD) subgroup in the same five questions 3, 6, 7, 11, and 12. The effect of the CSD on shortening average response time ($\bar{\mathbf{T}}_1$)

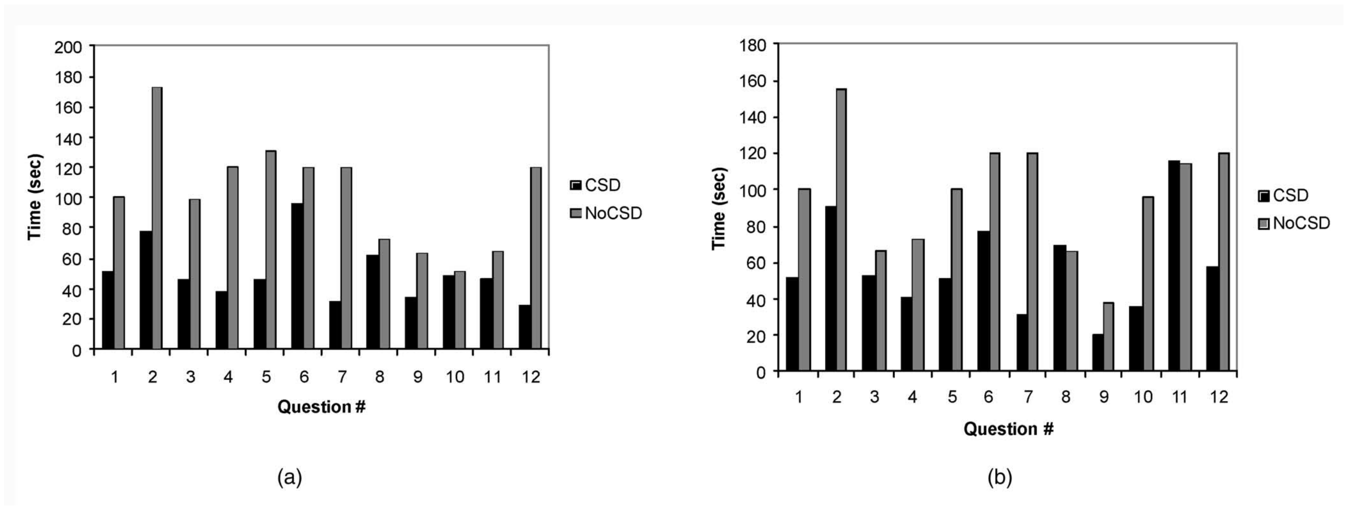


Fig. 5. Total average time taken to respond correctly. (a) Experiment I: the difference $\bar{\mathbf{T}}_2$ (NoCSD) - $\bar{\mathbf{T}}_2$ (CSD) = 52.42 was significant at the 0.0002 level. (b) Experiment II The difference $\bar{\mathbf{T}}_2$ (NoCSD) - $\bar{\mathbf{T}}_2$ (CSD) = 39.76 was significant at the 0.0035 level.

across all 12 questions, using a t-test, was significant at the 0.0216 level in Experiment I, but this positive effect was not significant in Experiment II. Similarly, the effect of the CSD on $\bar{\mathbf{T}}_1$, considering individual questions as another factor using **SAS (proc anova)**, was highly significant at the 0.0035 level, while its positive effect on $\bar{\mathbf{T}}_1$ was not significant in Experiment II (P-value = 0.1876).

4.4.2 Analysis of Average Correct Response Times ($\bar{\mathbf{T}}_2$)

Figs. 5a and 5b graph the average response times only for correct responses ($\bar{\mathbf{T}}_2$) over all 12 questions, where Fig. 5a pertains to the Experiment I and Fig. 5b pertains to Experiment II. Here, the CSD group consistently outperformed the control group and the positive effects of CSD, across all 12 questions using a t-test, were highly significant with P-value < 0.001 for Experiment I and P-value = 0.0035 for Experiment II. Further, when the questions were also considered as a factor in the experiment, **SAS (proc anova)**[2] showed that the effects of the CSD on $\bar{\mathbf{T}}_2$ were significant with P-values < 0.001 for both Experiment I and II. It should be noted that these results are strengthened by the fact that for three questions (6, 7, and 12) in both experiments there were no correct responses from the control group. The graphs in Fig. 5 select the average response times for the control group on those questions.

Before analyzing the data for the differences in number of correct responses, **X**, the obvious correlation between the random variables $\bar{\mathbf{T}}_2$ and $\bar{\mathbf{T}}_1$ had to be addressed. The random variables $\bar{\mathbf{T}}_2$ and $\bar{\mathbf{T}}_1$ are highly correlated because $\bar{\mathbf{T}}_2$ is simply the value of $\bar{\mathbf{T}}_1$ when the subject provided a correct answer to the corresponding question and, as a result, $\bar{\mathbf{T}}_2$ was the average of only correct response times, while $\bar{\mathbf{T}}_1$ was the mean of all the responses under a question type. The sample Pearson correlation coefficients between $\bar{\mathbf{T}}_2$ and $\bar{\mathbf{T}}_1$ were computed, by **SAS,** to be **r =** 0.7062 and 0.4606, respectively for Experiments I and II. These values were statistically significant for a right-tailed test at the 0.0000575 and 0.0118 probability levels. Although the value of Pearson correlation for Experiment II was less than 0.50, its t-statistic (with 22 degrees of freedom)

$$t_0 = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}} = 2.434$$

for testing H_0: p = 0 versus H_1: p > 0 was statistically significant at the 0.0118 level (see [26]).

A covariance analysis was performed in order to remove the effect of the concomitant variable $\bar{\mathbf{T}}_1$ on $\bar{\mathbf{T}}_2$. The **SAS** outputs in Figs. 6 and 7 show that, under Type III SS, the value of F statistic for the CSD from Experiment I was 10.65, which was statistically significant at the 0.37 percent level, and for Experiment II was 8.32, which was significant at the 0.89 percent level. This implies that without question the CSD had a statistically significant impact in helping the subjects to arrive at the correct answer regardless of the total time they spent to respond to a question.

The number of correct answers per question differed significantly under both CSD and NOCSD subgroups for both Experiments I and II. Indeed, the Chi-square statistic with 11 degrees of freedom for the CSD subgroup of Experiment I calculated (using a Matlab program) was $\chi_0^2 = 38.0207$, which was highly significant (P-value < 0.001). Therefore, the results of the unbalanced factorial experiments with CSD and Question Types as factors were analyzed using **SAS (proc glm).** Such analyses allowed the examination of interaction between the CSD and Question Type. The **SAS** outputs pertaining to Experiments I and II are provided in Figs. 8 and 9, respectively. Under Type III SS, the value of F statistics for the CSD of the Senior-class was 40.78 and that of the sophomore class was 24.54. These F_0 values are highly significant statistically, with P-values < 0.001. Appendices D and E, under Type III SS, also show that the effects of question type on $\bar{\mathbf{T}}_2$ were significant in both Experiments I and II (P-values = 0.0043 and 0.0002).

2. All SAS codes used in the analyses of both experiments are available on request through email to S. Maghsoodloo.

```
                         General Linear Models Procedure

                            Dependent Variable: T2

Source      DF      Sum of Squares          Mean Square      F0 Value      Pr > F

Model        2      22941287169.3007     11470643584.6503     21.07         0.0001

Error       21      11431359804.0325     544350466.858694

Total       23      34372646973.3333

R-Square         C.V.               Root MSE                T2 Mean

0.667429       30.37662          23331.31944102         76806.83333333


Source       DF         Type I SS            Mean Square        F0 Value     Pr > F

CSD          1      16489759504.1666     16489759504.1666       30.29       0.0001
T1           1      6451527665.13409     6451527665.13409       11.85       0.0024


Source       DF         Type III SS           Mean Square       F0 Value     Pr > F

CSD          1        5798325039.48273     5798325039.48273     10.65       0.0037
T1           1        6451527665.13409     6451527665.13409     11.85       0.0024


                                        T for H0:       Pr > |T|       Std Error of
Parameter   DF        Estimate         Parameter=0                       Estimate

INTERCEPT    1      18352.53193           1.59         0.1266         11535.85372
CSD          1      35140.62485           3.26         0.0037         10767.06912

T1 = Covariate          0.54796           3.44         0.0024             0.15917

* DF = Degrees of Freedom.
```

Fig. 6. The analysis of covariance for Experiment I using $\mathbf{T_1}$ as the covariate.

```
                         General Linear Models Procedure

                            Dependent Variable: T2

Source        DF        Sum of Squares          Mean Square        F0 Value      Pr > F

Model          2       12605816829.2928       6302908414.64644       8.11        0.0025

Error         21       16324052365.2071        777335826.91462

Total         23       28929869194.5000

R-Square                   C.V.            Root MSE              T2 Mean

0.435737                 35.95277       27880.74294051        77548.25000000


Source        DF         Type I SS            Mean Square          F0 Value      Pr > F

CSD           1      9486417962.66666      9486417962.66666000      12.20        0.0022
T1            1      3119398866.62622      3119398866.62622000       4.01        0.0582


Source        DF        Type III SS          Mean Square         F0 Value      Pr > F

CSD           1      6468552392.13576     6468552392.13576        8.32        0.0089
T1            1      3119398866.62622     3119398866.62622        4.01        0.0582


                                   T for H0:        Pr > |T|          Std Error of
Parameter             Estimate    Parameter=0                           Estimate

INTERCEPT     1     31589.25325       2.06          0.0516           15304.93637
CSD           1     33901.68621       2.88          0.0089           11752.27897
T1                      0.43419       2.00          0.0582               0.21674
```

Fig. 7. The analysis of covariance for Experiment II using $\mathbf{T_1}$ as the covariate.

```
                         General Linear Models Procedure

Dependent Variable: T2

Source        DF        Sum of Squares         Mean Square       F0 Value    Pr > F

Model         23       181072784154.379      7872729745.84259      5.20      0.0001

Error        140       212158899632.571      1515420711.66122

Total        163       393231683786.951

R-Square              C.V.                Root MSE              T2 Mean

0.460474            56.29950           38928.40494628        69145.20731707



Source        DF       Type I SS             Mean Square          F0 Value    Pr > F

CSD            1     94894341388.7551      94894341388.75510       62.62      0.0001
QTYPE         11     54461392059.8969       4951035641.80881        3.27      0.0005
CSD*QTYPE     11     31717050705.7274       2883368245.97522        1.90      0.0437



Source        DF        Type III SS            Mean Square        F0 Value    Pr > F

CSD            1      61797801660.3405      61797801660.3405       40.78      0.0001
QTYPE         11      44014812173.9986       4001346561.2726        2.64      0.0043
CSD*QTYPE     11      31717050705.7274       2883368245.9752        1.90      0.0437
```

Fig. 8. The factorial analysis of correct response times, T_2, from Experiment I.

```
                        General Linear Models Procedure

                            Dependent Variable: T2

Source        DF       Sum of Squares        Mean Square       F0 Value     Pr > F

Model         23      154007119238.836     6695961706.03636     5.01        0.0001

Error        163      217837942693.334     1336429096.27812

Corrected    186      371845061932.171

R-Square               C.V.              Root MSE              T2 Mean

0.414170             57.82092          36557.20306968        63224.87700535



Source        DF      Type I SS            Mean Square           F0 Value       Pr > F

CSD            1    70104969056.7829    70104969056.7829         52.46          0.0001
QTYPE         11    64699361772.1058     5881760161.1005          4.40          0.0001
CSD*QTYPE     11    19202788409.9478     1745708037.2680          1.31          0.2251



Source        DF       Type III SS           Mean Square          F0 Value     Pr > F

CSD            1     32795481187.7509     32795481187.75090        24.54       0.0001
QTYPE         11     51490586326.2791      4680962393.29810         3.50       0.0002
CSD*QTYPE     11     19202788409.9477      1745708037.26798         1.31       0.2251
```

Fig. 9. The factorial analysis of correct response times, T_2 factorial, from Experiment II.

However, the interaction effect between CSD and Question Type, CSD×Qtype, was significant only in the Experiment I at the 4.372 percent level. In summary, the SAS outputs in Appendices D and E clearly show the enormous impact of the CSD in aiding the subject to arrive at the correct response in shortest amount of time.

4.4.3 Analysis of the Number of Correct Responses (X)

Figs. 10a and 10b graph the total number of correct responses per question, **X**, for the two experiments. Again, the performance gains of the CSD subgroups are significant: For Experiment I, 45.18 percent of the CSD subgroup's responses were correct, while only 25.88

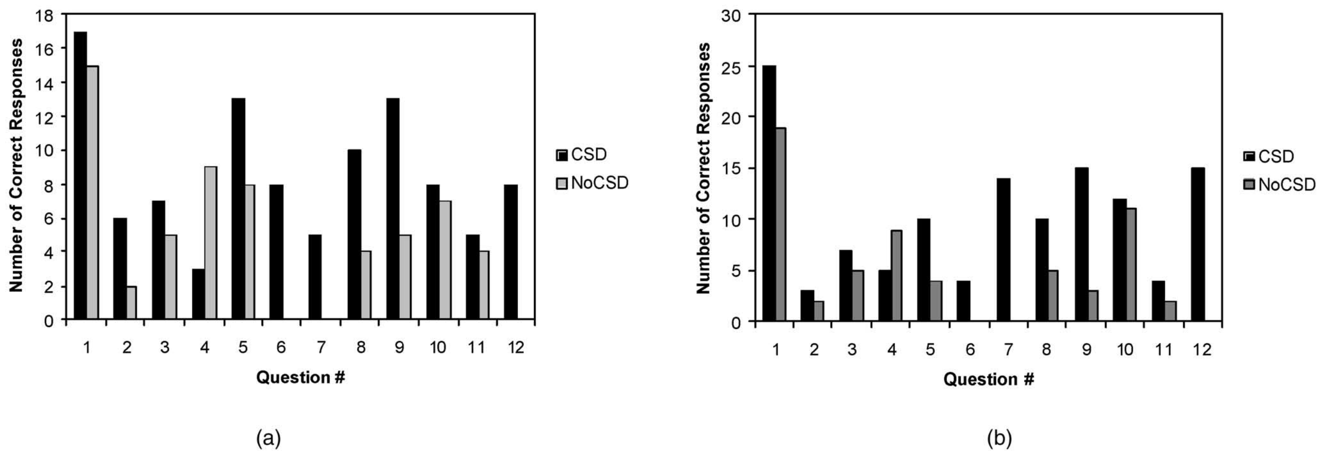


Fig. 10. Number of correct responses. (a) Experiment I: The difference $\hat{\mathbf{p}}_{\text{CSD}} - \hat{\mathbf{p}}_{\text{NOCSD}} = 0.1930$ was highly significant (P-value < 0.001). (b) Experiment II: The difference $\hat{\mathbf{p}}_{\text{CSD}} - \hat{\mathbf{p}}_{\text{NOCSD}} = 0.2133$ was highly significant (P-value < 0.001).

percent of the control subgroup's responses were correct. For Experiment II, 41.33 percent of the responses from the CSD subgroup were correct, while 20 percent of the responses from the NOCSD subgroup were correct, across all 12 questions. The standard errors of the difference between these two proportions, under the null hypothesis $H_0 : p_{\text{CSD}} = p_{\text{NOCSD}}$, were computed to be $se(\hat{\mathbf{p}}_{\text{CSD}} - \hat{\mathbf{p}}_{\text{NOCSD}}) = 0.0448$ and 0.0375 for Experiments I and II, respectively. The sample sizes used to compute these standard errors were $n_{\text{CSD}} = n_{\text{NOCSD}} = 19 \times 12 = 228$ for Experiment I, and $n = 12 \times 25 = 300$ for Experiment II. These led to a P-value = $2 \times P(\mathbf{Z} \geq 4.3053) < 0.001$ for Experiment I and a P-value = $2 \times P(\mathbf{Z} \geq 5.6662)$) < 0.001 for Experiment II. These very small P-values strongly reject the null hypothesis $H_0 : p_{\text{CSD}} = p_{\text{NOCSD}}$, implying that the differences between the two proportions of correct responses were statistically very significant for both experiments.

Comparisons were also made between the Senior (Experiment I) and Sophomore (Experiment II) classes with regard to the variable $\bar{\mathbf{T}}_1$ (CSD_{I} average time versus CSD_{II}, and NOCSD_{I} versus NOCSD_{II}), variable $\bar{\mathbf{T}}_2$ (CSD_{I} versus CSD_{II}, and NOCSD_{I} versus NOCSD_{II}), and $\mathbf{X}_{\mathbf{I}}$ versus $\mathbf{X}_{\mathbf{II}}$. However, none of the differences were found to be statistically or practically significant, implying that experience level had no significant impact on the utility of CSD in helping the reader comprehend a program control flow structure more readily.

5 Conclusions and Evaluation

Several conclusions can be drawn from the above results. A fundamental conclusion that is obvious from an analysis of the results is that the CSD had a highly significant positive effect on subject performance during the comprehension activity. Other conclusions, though, suggest not only strengths of the CSD, but also possible limitations of both the visualization as well as the experimental method employed.

The positive effect of the CSD on subject performance was the least strong for the $\bar{\mathbf{T}}_1$ measure (average time to respond), with statistically significant results only in one of the two experiments. However, since $\bar{\mathbf{T}}_1$ is only a measure of time without regard to correctness, it shouldn't be used in isolation. Correctness is more important than time and, thus, must always be taken into account when interpreting results. For example, on the questions where the control group outperformed the CSD group with respect to $\bar{\mathbf{T}}_1$, that is where the control group responded more quickly on average than the CSD group, the control group was much less likely to answer correctly.

Interpretation of the results regarding $\bar{\mathbf{T}}_2$ (average time taken to respond correctly) are much more useful than those regarding $\bar{\mathbf{T}}_1$ since $\bar{\mathbf{T}}_2$ is a measure of time with respect to correctness. When correctness is considered, the CSD group consistently responded more quickly than the control group, with highly significant statistical differences. There are only two exceptions to this. In Experiment 2, the control group performed slightly better than the CSD group on questions 8 and 11. However, this must be considered in light of the fact that the CSD group was roughly twice as likely as the control group to answer those two questions correctly.

When correctness was considered independently of time, as in the $\mathbf{X}$ measure, the CSD again consistently outperformed the control group, with high statistical significance. In both experiments, the CSD group averaged almost twice as many correct responses as the control group. This result, along with the covariance analysis of the effect of $\bar{\mathbf{T}}_1$ on $\bar{\mathbf{T}}_2$, implies that the CSD had a highly significant effect in helping the subjects respond correctly to the questions, regardless of the total time taken to respond.

These strongly positive effects of the CSD on subject performance can be ascribed to fundamental attributes of the diagram. The source code used in the experiment contained relatively large control structures that contained several levels of nesting and spanned many lines, thus decreasing the visibility of control structures and control flow paths. Since the CSD clearly delineated each control structure and provided context and continuity for the nested constructs, comprehension efficiency was increased. Other studies [10], [5] have found that good typography and other semantic cues can increase comprehension. Since the CSD adds visual cues for control constructs in addition to existing typographic conventions and familiar indentation, the diagram can facilitate comprehension processes and, thus, demonstrate superior performance to plain text.

An examination of the data from both experiments with respect to $\bar{\mathbf{T}}_1$ and $\bar{\mathbf{T}}_2$ (Figs. 4 and 5) reveals that the performance gap between the control group and the CSD group narrows over the 12 questions. That is, the performance gains of the CSD are more pronounced in earlier questions than in later questions. Since the order of the questions was random and since there are instances of the same question type distributed evenly throughout the order (see Section 2.2), this effect cannot be attributed to the questions themselves. The observed trend must be a learning effect that accumulates as the experiments proceed, that is, as the subjects become more familiar and comfortable with the experimental task. This suggests that the beneficial effect of the CSD with respect to response speed may in fact decrease with task repetition. It is important to note, however, that this trend does not exist in the $\mathbf{X}$ measure (Fig. 10). That is, the beneficial effect of the CSD with respect to response correctness is persistent and does not diminish with task repetition. Further experimentation is needed to completely describe and explain this effect, however.

A close examination of the results relative to the $\mathbf{X}$ measure reveals a possible limitation of the CSD. Considering question 4 in isolation, the CSD group performed roughly two to three times better than the control group with respect to time ($\bar{\mathbf{T}}_1$ and $\bar{\mathbf{T}}_2$), but with approximately one half of the correctness ($\mathbf{X}$). That is, the CSD group was on average twice as fast as the control group on question 4, but responded with twice as many incorrect answers. The increased average speed suggests that the CSD group was confident of their incorrect answers. This immediately raises the concern that the CSD was misleading the subjects to make an incorrect response for this question. Question 4 asked the subjects to enumerate the loops present in the source code. One of the loops was a for-loop written completely on one line and, as a result, its CSD representation was not as visually distinct as a multiline loop. Subjects in the CSD group were most likely scanning the source code for the CSD loop symbol (see Appendix A) and were thus mislead by the single line loop that did not appear to have the same CSD symbol as the multiline loops. Subjects in the control group were most likely scanning the source code for the various loop keywords (for, while, do) and were thus not mislead by a loop that appeared on a single line. While this is a potential shortcoming of the CSD, the effect is likely to be observed in relatively rare situations and is thus not a major limitation of the diagram.

There are also limitations of the data analysis and experimental method that must be addressed. $\bar{\mathbf{T}}_1$ was averaged over correct and incorrect responses. It has been suggested that distinct cognitive processes occur in each of these conditions. The data analysis did not try to measure this effect or account for it in any way. Thus, this is a possible limitation of the results involving $\bar{\mathbf{T}}_1$.

TABLE 1
Summary of Experimental Results

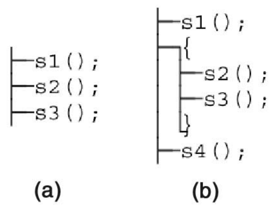
		Experiment I	Experiment II
		P-value	
$\bar{T}_1$	Over all questions	0.0216	> 0.05
	Question as a factor	0.0035	> 0.05
$\bar{T}_2$	Over all questions	< 0.001	0.0035
	Question as a factor	< 0.001	< 0.001
X	Difference in proportion of correct responses	< 0.001	< 0.001

Other studies have shown that individual differences among subjects can account for a large amount of the variance in performance. The experimental design addressed this by ensuring that the control group and the CSD group were performance balanced with respect to each other. However, a repeated measures design is perhaps more appropriate and, thus, its absence could be considered a limitation.

6 Summary and Future Work

This work has directly addressed the effectiveness of software visualization in comprehension tasks. Specifically, the CSD was shown to have a highly significant positive effect on subject performance in experimental tasks, from the standpoint of both shortening response times and increasing correctness. Table 1 summarizes the statistical significance of the results with respect to average time to respond ($\bar{\mathbf{T}}_1$), average time to respond correctly ($\bar{\mathbf{T}}_2$), and number of correct responses ($\mathbf{X}$). The practical significance of these results is strengthened by the fact that the experimental task focused a code reading and comprehension activity that is common to most, if not all, source code comprehension tasks and most cognitive models of program comprehension. Thus, these results should be applicable to a variety of specific software engineering contexts and tasks.

Although the positive effect of the CSD on $\bar{\mathbf{T}}_1$ was statistically significant in Experiment I but not in Experiment II, this particular measure is not likely to be of practical importance. Increased speed is desirable only when paired with increased (or at least no worse) correctness. Thus, the measures of greatest interest are $\bar{\mathbf{T}}_2$ and $\mathbf{X}$. In each case, the CSD was shown to have a highly significant positive effect on subject performance. This positive impact remained highly significant even when removing the effect of the concomitant variable $\bar{\mathbf{T}}_1$ on $\bar{\mathbf{T}}_2$.



```
s1();
s2();
s3();
```

```
s1();
{
  s2();
  s3();
}
s4();
```

Fig. 11. CSD symbols for sequential control.

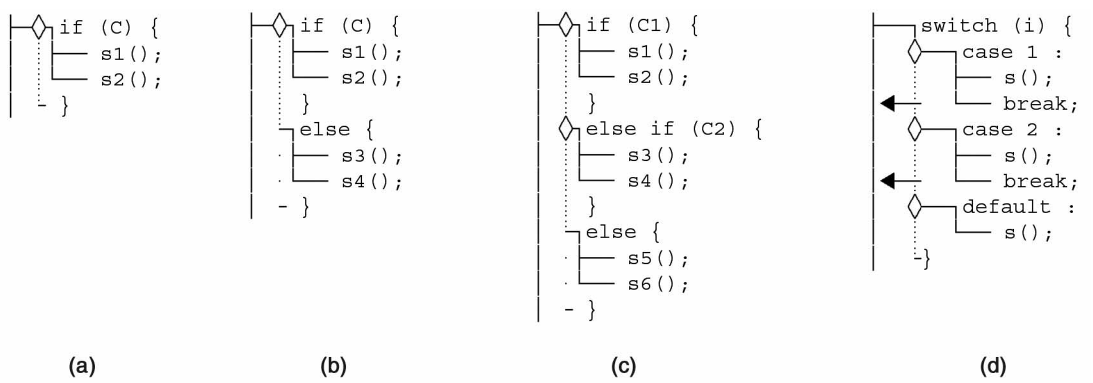
Indeed, the result with perhaps the greatest practical importance and the most relevance to current software engineering research and practice is that the CSD had a significant impact on the subjects' ability to arrive at the correct answer to a question, regardless of the length of time taken to respond. This suggests that the CSD could be a valuable tool in code reading tasks. Since code reading and comprehension activities are so widespread in the software life cycle, the CSD has the potential to be a valuable aid to software professionals. Indeed, Basili [7] has acknowledged code reading as a fundamental, crucial technology for achieving quality software, and the only analysis technology that can be applied throughout the software life cycle. Other research has also suggested that increasing the effectiveness of code reading could have a positive impact on the effectiveness of life cycle activities such as software inspections [35].

Since the subjects in Experiment I were seniors and graduate students, reasonably expert in the experimental task, and the subjects in Experiment II were sophomores, and certainly novices at the experimental task, the issue of novice versus expert performance could also be addressed. Existing research on this issue with respect to software visualization has produced conflicting results and inferences. Some researchers have suggested, in personal correspondence, that novices rather than experts should achieve a greater performance gain with the CSD since it would help offset their unfamiliarity with the conventional (plain text) representation. Other research [31] suggests that the increased graphical readership skills of experts would allow that group to enjoy a greater performance gain that novices. A comparison of the groups in Experiments I and II, however, suggest that the reader's experience level had no significant impact on the utility of the CSD in increasing performance in the experimental task.

Follow-up experiments are planned within a larger research program that will build on these results and will further explore the human performance benefits offered by certain software visualizations. Specifically, experiments are planned to assess the possible benefit afforded by the CSD in the context of particular reading techniques and technologies [7] and in particular tasks such as software inspections performed by practicing software professionals [35].

A robust experimental framework is needed to measure the impact of software visualizations on the overall software process in terms of productivity and quality, thus, directly addressing the open question as to the relative effectiveness of visualizations on production software. This is an important research question since the utility of software visualization has yet to be convincingly demonstrated in an industrial setting. An experimental evaluation of visualizations such as the CSD within a best practices development environment on production software should stimulate organizations to modify their software development processes to include those visualizations which, based on the experimental results, indicate the potential for substantial gains in productivity and reliability.

Continued empirical research is clearly needed. While this work has shown promising results, it is crucial that the research be systematically scaled and extended into large-scale industrial settings. A robust experimental research program, building directly on this work, will allow researchers to address many open questions and thereby



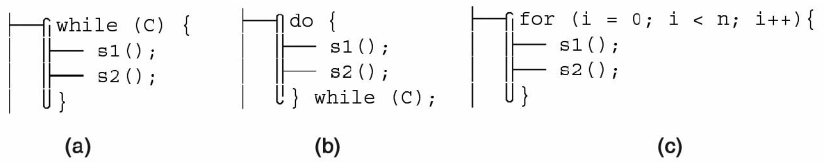
```
if (C) {
    s1();
    s2();
}
```

```
if (C) {
    s1();
    s2();
}
else {
    s3();
    s4();
}
```

```
if (C1) {
    s1();
    s2();
}
else if (C2) {
    s3();
    s4();
}
else {
    s5();
    s6();
}
```

```
switch (i) {
    case 1 :
        s();
        break;
    case 2 :
        s();
        break;
    default :
        s();
}
```

Fig. 12. CSD symbols for selection constructs.

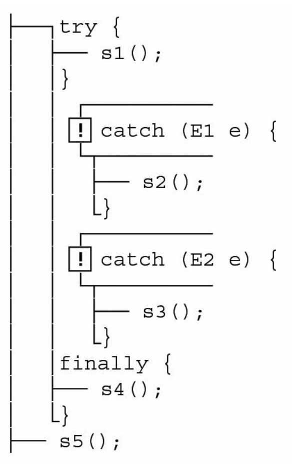


```
while (C) {
    s1();
    s2();
}
```

```
do {
    s1();
    s2();
} while (C);
```

```
for (i = 0; i < n; i++){
    s1();
    s2();
}
```

Fig. 13. CSD symbols for iteration constructs.



```
try {
   s1();
}
catch (E1 e) {
   s2();
}
catch (E2 e) {
   s3();
}
finally {
   s4();
}
s5();
```

Fig. 14. CSD symbols for exception handling.

make significant contributions to the software engineering community.

Appendix A

The Control Structure Diagram for Java

The CSD provides graphical representations for all major control constructs in Java, as well as Ada 95, C, and C++, including sequence, selection, iteration, and exception handling. Data declarations as well as module-level constructs such as classes and methods also have representations in the CSD. This appendix illustrates the major CSD elements for Java, as used in the experiments reported in this paper. The GRASP family of editors can be used to automatically generate CSDs for any of the supported languages. GRASP editors are available for download at http://www.eng.auburn.edu/grasp.

Sequential control is the most basic, and its graphical symbols appear in the CSD for most other control constructs. The individual statements in sequence are located at the end of horizontal stems hung from a vertical spine (⊢). There is one vertical spine for each syntactical level of sequential control. Figs. 11a and 11b illustrate purely sequential CSDs. Notice that the compound statement in Fig. 11b begins a new syntactical level and, thus, adds a vertical spine to the CSD.

The diamond symbol (◇) indicates a point of decision in the control flow, that is, a selection control structure. A solid stem (–) from the right side of the diamond indicates the control path to be followed when the condition is true, while a dashed spine (⁝) from the bottom of the diamond

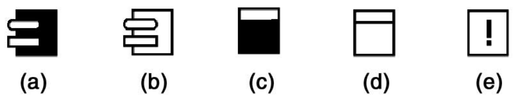


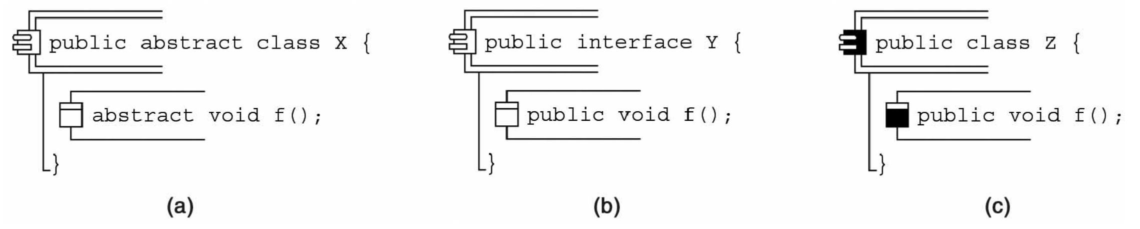
Fig. 15. CSD unit symbols.

indicates the alternate false path. Notice that the sequential CSD forms are, of course, present within the selection structures. Figs. 12a, 12b, 12c, and 12d illustrate the four basic forms of selection in Java. The switch statement in Fig. 12d employs left arrows (←) and right arrows (◇—|) to indicate immediate transfers of control; the left arrow in the case of a break statement that ends execution in the switch and the right arrow in the case of the immediate jump to the appropriate case clause (as opposed to the top-to-bottom evaluation semantics of the if-else-if).

Iterative statements are depicted by a graphic symbol that loops upon itself but is broken at the syntactic point at which control may exit the loop. Thus, the pre-test while and for loop CSDs in Figs. 13a, 13b, and 13c, respectively, are broken at the top and the posttest do loop CSD in Fig. 13b is broken at the bottom. The vertical length of the loop CSD indicates the syntactic extent of the iterative statement.

Exception handlers in Java are implemented with try-catch-finally statements. Statements in the try block are executed sequentially until an exception is raised, at which point control is immediately transferred to at most one of the associated catch blocks. If none of the catch blocks can handle the exception, control propagates back to the calling unit. After either 1) the try block exits normally, or 2) a local catch block is executed, control transfers to the finally block. Obviously, much of this exception handling semantics is dynamic and thus cannot be represented in a static visualization such as the CSD. However, the CSD does convey the static control relationships among the try, catch, and finally blocks. Notice in Fig. 14 that the try and finally blocks are represented as sequential CSDs attached to the same vertical spine. The catch blocks are not attached to the vertical spine, indicating that they are not part of the normal sequential flow, and they are further distinguished by the exception unit symbol ().

Module-level constructs such as classes, interfaces, and methods are each represented by a combination of unit symbols, horizontal boxes, and vertical spines. Unit symbols, based on [9] and shown in Fig. 15, are used to indicate the type of construct present. Optional horizontal boxes, both double- and single-width, are used to visually delimit the header or specification portion of the construct.

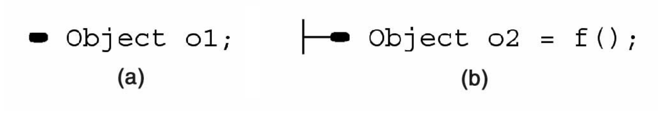


```
public abstract class X {
   abstract void f();
}
```

```
public interface Y {
   public void f();
}
```

```
public class Z {
   public void f();
}
```

Fig. 16. CSD symbols for classes and methods.



```
Object o1;
Object o2 = f();
```

Fig. 17. CSD data declaration symbol.

A vertical spine visually delimits the syntactical scope of the construct. The CSDs (with optional box notation) for classes, both concrete and abstract, as well as interfaces are illustrated in Figs. 16a, 16b, and 16c.

Data declarations also have graphical representations in the CSD. Fig. 17a illustrates a simple declaration of a reference variable in a declarative section of a Java module. Since statements are illegal in declarative section, there is no stem or spine to indicate control flow. Only the data symbol (●) is present. Java allows variables to be declared in statement sections as well as declarative sections, however. Fig. 17b illustrates the declaration of an object as part of an executable statement. Since this declaration is part of a control structure, the data symbol is attached to a stem and spine ().

Acknowledgments

This work was supported, in part, by a grant from the US National Science Foundation (EIA-9806777).

References

[1] M. Aoyama, "Design Specification in Japan: Tree-Structured Charts," *IEEE Software,* pp. 31-37, Mar. 1989.

[2] V. Arunachalam and W. Sasso, "Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Engineering," *J. Systems and Software,* vol. 34, pp. 177-189, 1996.

[3] A. Badre, "Designing Chunks for Sequentially Displayed Information," *Directions in Human Computer Interaction* A. Badre and B. Shneiderman, eds., pp. 179-193, 1982.

[4] R.M. Baecker, C. Digiano, and A. Marcus, "Software Visualization for Debugging," *Comm. ACM,* vol. 40, no. 4, pp. 44-54, 1997.

[5] R.M. Baecker and A. Marcus, *Human Factors and Typography For More Readable Programs.* ACM Press, 1990.

[6] T. Ball and S.G. Eick, "Software Visualization in the Large," *Computer,* pp. 33-43, Apr. 1996.

[7] V. Basili, "Evolving and Packaging Reading Technologies," *J. Systems and Software,* vol. 38, pp. 3-12, 1997.

[8] V. Basili and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.,* vol. 13, no. 12, pp. 1278-1296, Dec. 1987.

[9] G. Booch and D. Bryan, *Software Engineering with Ada,* third ed. Benjamin-Cummings, 1994.

[10] D.G. Bouwhuis, "Reading as a Goal-Driven Behaviour," *Working Models of Human Perception,* B.A.G. Elsendoorn and H. Bouma, eds., pp. 341-362, 1988.

[11] S.N. Cant, D.R. Jeffery, and B. Henderson-Sellers, "A Conceptual Model of Cognitive Complexity of Elements of the Programming Process," *Information and Software Technology,* vol. 37, no. 7, pp. 351-362, 1995.

[12] W. Citrin, C. Santiago, and B. Zorn, "Scalable Interfaces to Support Program Comprehension," *Proc. Fourth Workshop Program Comprehension,* pp. 123-132, 1996.

[13] T.A. Corbi, "Program Understanding: Challenge for the 1990s," *IBM Systems J.,* vol. 28, no. 2 pp. 294-306, 1989.

[14] J.H. Cross, T.D. Hendrix, L.A. Barowski, and K.S. Mathias, "Scalable Visualizations to Support Reverse Engineering: A Framework for Evaluation," *Proc. Fifth Working Conf. Reverse Eng.,* Oct. 1998.

[15] J.H. Cross, S. Maghsoodloo, and T.D. Hendrix, "The Control Structure Diagram: An Overview and Initial Evaluation," *Empirical Software Eng.,* vol. 3, no. 2, pp. 131-156, 1998.

[16] J.H. Cross and S.V. Sheppard, "The Control Structure Diagram: An Automated Graphical Representation for Software," *Proc. 21st Hawaii Int'l Conf. System Sciences,* vol. 2, pp. 446-454, 1988.

[17] B. Curtis, S.B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D.A. Boehm-Davis, "Experimental Evaluation of Software Documentation Formats," *J. Systems and Software,* vol. 9, pp. 167-207, 1989.

[18] J.S. Davis, "Chunks: A Basis for Complexity Measurement," *Information Processing and Management,* vol. 20, no. 1, pp. 119-127, 1984.

[19] P. Goolkasian, "Picture-Word Differences in a Sentence Verification Task," *Memory & Cognition,* vol. 24, pp. 584-594, 1996.

[20] T.R.G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs," *Proc. Sixth European Conf. Cognitive Ergonomics (ECCE-6),* 1992.

[21] T.R.G. Green, M. Petre, and R.K.E. Bellamy, "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture," *Empirical Studies of Programmers Fourth Workshop,* 1991.

[22] W. Kintsch, *Memory and Cognition.* John Wiley, 1977.

[23] J.R. Miara, J.A. Musselman, J.A. Navarro, and B. Shneiderman, "Program Indentation and Comprehensibility," *Comm. ACM,* vol. 26, no. 11, pp. 861-867, 1983.

[24] G.A. Miller, "The Magic Seven Plus or Minus Two. Some Limits on Our Capacity for Processing Information," *Psychological Re.,* vol. 63, pp. 81-97, 1956.

[25] T.G. Moher, D.C. Mak, B. Blumenthal, and L.M. Leventhal, "Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets," *Empirical Studies of Programmers: Fifth Workshop,* C.R. Cook, J.C. Scholtz, and J.C. Spohrer, eds., pp. 137-161, 1993.

[26] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers,* second ed. pp. 473-475, 1999.

[27] G.C. Murphy, D. Notkin, and E. Lan, "An Empirical Study of Static Call Graph Extractors," *Proc. 18th Int'l Conf. Software Eng.,* Mar. 1996.

[28] P.W. Oman and C.R. Cook, "Typographic Style is More than Cosmetic," *Comm. ACM,* vol. 33, no. 5, pp. 506-520, 1990.

[29] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology,* vol. 19, pp. 295-341, 1987.

[30] N. Pennington, "Comprehension Strategies in Programming," *Proc. Empirical Studies of Programmers: Second Workshop,* pp. 100-112, 1987.

[31] M. Petre, "Why Looking isn't Always Seeing: Readership Skills and Graphical Programming," *Comm. ACM,* vol. 38, no. 6, pp. 33-44, 1995.

[32] B.A. Price, R.M. Baecker, and I.S. Small, "A Principled Taxonomy Of Software Visualization," *J. Visual Languages and Computing,* vol. 4, no. 3 pp. 211-266, 1993.

[33] Rational *UML Summary,* version 1.01, Rational Software Corporation, Santa Clara, Calif., 1997.

[34] D. Raymond, "Characterizing Visual Languages," *Proc. 1991 IEEE Workshop Visual Languages,* pp. 176-182, 1991.

[35] C. Sauer, D.R. Jeffery, L. Land, and P. Yetton, "The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 1-14, 2000.

[36] D.A. Scanlan, "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison," *IEEE Software,* pp. 28-36, Sept. 1989.

[37] B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *Int'l J. Computer and Information Sciences,* vol. 8, no. 3, pp. 219-238, 1979.

[38] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.,* vol. 10, no. 5, pp. 595-609, 1984.

[39] M.D. Storey, F.D. Fracchia, and H.A. Muller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization," *Proc. Fifth Int'l Workshop Program Comprehension (IWPC'97),* pp. 17-28, Mar. 1997.

[40] A. von Mayrhauser and A.M. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment," *Proc. Second Workshop Program Comprehension,* pp. 55-63, July 1993.

[41] A. von Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 424-437, 1996.
[42] M. Weiser, "Source Code," *Computer*, vol. 20, no. 11, pp. 66-73, Nov. 1987.
[43] M.S. Weldon, H.L. Roediger, and B.H. Challis, "The Properties of Retrieval Cues Constrain the Picture Superiority Effect," *Memory and Cognition*, vol. 17, pp. 95-105, 1989.

Dean Hendrix is an associate professor of computer science and software engineering at Auburn University, Alabama. His research interests are focused primarily within software engineering and address two major areas of work: software visualization and software process. The overall goal of his continued research is to effect fundamental improvements in software engineering best practices. Dr. Hendrix is a member of the ACM, the IEEE, and the IEEE Computer Society.

James H. Cross II is professor and chair of the Computer Science and Software Engineering Department at Auburn University, Alabama. His primary interests are teaching undergraduate and graduate courses in software engineering and directing research in the areas of software methodology, testing, and reverse engineering. His continuing research efforts include the GRASP project, which focuses on the reverse engineering and the automatic generation of graphical representations of software. Dr. Cross is a member of the ACM and a senior member of the IEEE and member of the Computer Society, of which he currently serves as vice president of chapter activities, cochair of Computing Curricula 2001 (CC2001), and as a member of the Executive Committee. Dr. Cross also serves on the CSAB Board of Directors.

Saeed Maghsoodloo is a professor of industrial and systems engineering at Auburn University, Alabama. His research interests and publications are in nonparametric statistics, multivariate analysis, experimental design and Taguchi methods, regression correlation/time-series analysis, quality control, and reliability engineering. In addition to being an associate editor of *IIE Transactions* and a registered professional engineer, he is a member of the ASA and ASQC.