Mental models and computer programming

José Juan Cañas†, María Teresa Bajo and Pilar Gonzalvo
Departamento de Psicología Experimental y Fisiología del Comportamiento,
Universidad de Granada, Campus de Cartuja, 18071 Granada, Spain

(Received 29 October 1992 and accepted in revised form 1 March 1994)

Programming is a cognitive activity that requires the learning of new reasoning skills and the understanding of new technical information. Since novices lack domain-specific knowledge, many instructional techniques attempt to provide them with a framework or mental model that can be used for incorporating new information. A major research question concerns how to encourage the acquisition of good mental models and how these models influence the learning process. One possible technique for providing an effective mental model is to use dynamic cues that make transparent to the user all the changes in the variable values, source codes, output, etc., as the program runs. Two groups of novice programmers were used in the experiment. All subjects learned some programming notions in the C language (MIXC). The MIXC version of the programming language provides a debugging facility (C trace) designed to show through a system window all the program components. Subjects were either allowed to use this facility or not allowed to do so. Performance measures of programming and debugging were taken as well as measures directed to assess subjects' mental models. Results showed differences in the way in which the two groups represented and organized programming concepts, although the performance tasks did not show parallel effects.

Introduction

It is well known that computer programming is a difficult but useful skill to learn. Recently, there has been a large body of research directed toward finding effective ways of facilitating the acquisition of this skill. One approach has focused on studying those factors that increase the novice understanding of computer programming. For example, it is widely accepted that programming requires having access to some sort of “mental model” of the system (Mayer, 1981; Young, 1981, 1983). Therefore, part of the research efforts in this area has been directed toward enhancing novices’ “mental models”. A “mental model” refers to the user’s mental representation of the components and operating rules of the system and may vary with respect to its completeness and veridicality (Mayer, 1981). Thus, a mental model might contain more or less details of the components and rules of the system and may match more or less closely the real characteristics of the system. Obviously, the more complete and veridical the user’s mental model is, the more useful in supporting sophisticated programming it will be.

Mayer (1981) studied the mental model acquired by students as they were learning to program in Basic. He describes Basic statements as a list of transactions consisting of an operation on some object at some location in the computer. These

† For reprint requests.
transactions usually involve a knowledge of how different components of the computer (input system, memory, program, output system, etc.) interact to produce a specific action. A good mental model would contain a good understanding of those transactions. However, the main difficulty for acquiring a good mental model is that those transactions involve operations, objects and locations that are not visible to the novice programmer. Therefore, a possible way of facilitating the acquisition of this mental model is to make visible to the user the internal states and operations of the system. Mayer (1981) provided “mental model training” to novice programmers by providing them with pictures representing the different components of the computer. These pictures were presented in a manual that specified how different Basic transactions made use of the components and operating rules of the computer. His results showed that mental model training increased subjects’ performance in Basic problems, especially for low-ability subjects. Similarly, Perkins, Schwartz and Simmons (1988) introduced a programming metacourse to novices learning Basic programming. The metacourse was designed to help students learn a visual model of what happens inside the computer. Students received forms displaying in a graphic format the variables and their values, flow of control and characters on the screen so that they could mentally represent the effects of any given command. Again, results indicated that subjects receiving this metacourse improved their programming and debugging performance relative to the performance of a control group.

Other studies have obtained similar encouraging results (du Boulay & O’Shea, 1981; Mayer, 1989). Hence, a large amount of effort has been dedicated to creating programming environments that make visible to the user the operations of the computer (du Boulay, O’Shea & Monk, 1989; Eisenstadt & Brayshaw, 1989). These environments usually include dynamic components of programming such as the sequence of events as they occur in the program or the particular circumstances that lead the program to a given state. For example, some languages have been implemented so that either pictorial or written traces can be displayed showing the actions and operations taken as the program runs. However, much of the empirical research has focused on more static information, such as that provided by concrete models, manuals, or static pictures, and very little research has been spent in testing the efficacy of programming tracers or similar environments. An aim of this paper is to test the efficacy of one such environment.

In addition, although the studies discussed above were intended to enhance novices’ mental models, they did not offer any assessment of the changes in subjects’ mental representations as they learned. Recent research examining the conceptual structures of novice and expert programmers has indicated that experts not only possess more programming knowledge than novices, but also organize this knowledge differently. For example, Adelson (1981) had expert and novice programmers recall lines of a program. A multidimensional scaling analysis of conceptual proximity derived from subjects’ recall order showed that experts organized the programming lines semantically according to programs and routines, whereas novices used a syntax-based organization according to surface features of the programming lines. Similarly, Cooke and Schvaneveldt (1988) had subjects who varied in degree of computer programming experience (naive, novice, intermediate and advanced programmers) make relatedness ratings on pairs of programming concepts. These ratings were analysed using the “Pathfinder scaling algorithm” and
showed that the four groups differed in the way that the concepts were represented, with more misdefined concepts as programming experience decreased.

Therefore, it is possible to assess subjects' mental representation or mental models using tasks such as recall, categorization or relatedness judgments. These tasks seem sensitive to changes in those representations as the students learn to program. In the present study two groups of naive programmers learned the C computer programming language with or without a tracer. The version of the C programming language selected for this study was the MIXC compiler. MIXC includes a facility designed to show the user the sequence of programming lines as they are executed by the program and the values that the different variables take as the program runs. Subjects’ mental representations of C programming concepts and their ability to write and debug simple programs (performance measures) were assessed at different stages of learning. It was expected that subjects’ performance in the programming and debugging tasks and subjects’ mental models would change and improve as a function of experience. Two assessment sessions were introduced to capture the effect of degree of programming experience. In addition, we expected that subjects using the Trace facility would show different mental representations than the Non-Trace subjects, and that these differences in the way they structured their programming knowledge would be paralleled in their programming and debugging skills.

**Method**

**SUBJECTS**

Two groups of four subjects formed the two between subjects conditions. Four subjects were assigned to the Trace condition and four to the Non-Trace condition. All subjects were in their third year of psychology and did not have any previous computer experience.

**PROCEDURE**

All subjects participated in two experimental phases in which they learned to program in the C programming language and were then tested on it. The first phase lasted approximately six months during which subjects received a weekly class in C. The purpose of the classes offered during this first phase was to introduce the subjects to the main concepts of programming in C. Therefore, the experimenter explained a new concept during the first hour of the class and then the subjects worked approximately for another hour with a simple program that exemplified that concept. For example, in one session the experimenter explained the concept “type of data”, including explanations of their different types, when to use them, the syntax involved, etc. In the second half of the session, subjects were introduced to an example of a program that included these “different types of data” and ran it several times until they completely understood what the program was doing. This training procedure was chosen to simulate typical programming classes in the computer programming domain.

†MIXC compiler is a trademark of MIX Software, Inc.
The Trace and Non-Trace groups were in separate classes. Great care was taken to repeat the classes in the same manner for both groups, except that the Trace group was allowed to use the trace facility, and the Non-Trace group was not. During the time between any two classes subjects were assigned a reading from the C manual by Waite, Prata and Martin (1984). After this first phase, subjects were assessed in their ability to write a simple program, debug a program error and in their knowledge of the relations among the main concepts of C.

Two months after the end of the first assessment session, a second phase started with the main purpose of teaching subjects to write their own programs. This second phase was also introduced with the purpose of simulating typical practical sessions and homework offered to students in computer courses. During this phase, the experimenter did not introduce new concepts, but supervised the students while they wrote their programs. The main difference from the first phase was that subjects did not work with already written programs, but were given simple problems and tried to write their own programs to solve them. Again, the Trace and Non-Trace groups were in separate but equivalent classes that only differed in the use of the trace facility. This phase lasted approximately two months, and as the sessions progressed subjects wrote programs of increasing difficulty. After this phase, subjects were again assessed on their programming abilities.

The assessment sessions for both phases had three parts. First, subjects completed a relatedness task concerning 14 C concepts. Subjects were seated in front of an IBM microcomputer and read instructions describing the task. Subjects were to assign ratings to pairs of concepts according to how related they thought the concepts were. The scale ranged from 0–9. A rating of zero indicated that the pair were very unrelated, and a rating of nine indicated a high degree of relatedness. The subjects were to indicate their responses by pressing the numbers corresponding to their ratings on the keyboard. The instructions emphasized that they should work fast, basing their ratings on their first impression of relatedness.

After the instructions were read the relatedness task began. All possible pairs of 14 concepts (91 pairs) were presented one at a time in a random order. The order of presentation within a pair was random and constant across subjects. That is, once the pairs were formed, one item always appeared in the first position and the other item always appeared in the second position. Each pair remained on the screen until the subject responded.

Second, subjects were given instructions to write a simple program and to explain aloud every thought that came to their minds as they were performing the task. The instructions gave special emphasis to the importance of talking aloud, and the experimenter reminded the subjects to do so periodically throughout the session. The program for the first phase consisted of writing functions that counted the number of digits, the number of letters and the number of blanks as they were entered on the keyboard. The program should have three functions that each returned a truth value depending on whether the character entered was a blank, a digit or a letter. The program for the second phase consisted of calculating the difference in hours and minutes between two times introduced from the keyboard in a format of minutes. Subjects were instructed to write a function that transformed the two times from minutes to hours and minutes and returned the result to the main function which would find the difference between the two times. They had two
hours to perform the task and they could use their books and notes as much as they wanted during the session. Subjects were video-taped while they were working. A second video camera was also used to record the computer screen during the whole session.

Third, subjects were presented with simple programs containing a semantic error and were instructed to find the error and correct the programs so that they would perform the task specified by the instructions. In the first phase of the experiment a single program was presented to the subjects on the computer screen, and the subjects could run and change it as many times as they wanted. In the second phase two programs were presented: one on the computer screen and one on paper. The idea behind this manipulation is that those subjects having a better mental model should also be better at mentally following the program's instructions. Therefore, the differences between the Trace and Non-Trace groups would be more evident in the paper condition. The order of the paper and screen conditions was balanced so that half of the subjects in each group had the program on paper first and the program on the screen second, and the other half had the reverse order. Assignment of programs to conditions was also balanced so that half of the time a program was assigned to the paper condition and the remaining time it was assigned to the screen condition. In both phases, subjects were instructed to verbalize what they were doing and thinking while performing the task. Both the subject and the computer screen were video-taped in both phases.

Results

RATING TASK

A distance matrix was constructed for each subject at each phase. Relatedness ratings were converted to distance by subtraction from nine. These distance matrices were then submitted to a Weighted Multidimensional Scaling Procedure (Schiffman, Reynolds & Young, 1981). The weighted procedure has the advantage of accepting more than one distance matrix. As with unweighted procedures, this algorithm first locates all the points representing the concepts in the n-dimensional space considering all the subjects as a group. Second, it estimates the particular contribution of each matrix to the multidimensional solution and assigns a particular weight to each subject on each dimension. These subjects' weights are represented in the space by vectors drawn from the origin of the space. Hence, the weighted procedure offers two outputs: (1) an n-dimensional space containing the conceptual representation that is common to all subjects, and (2) an n-dimensional space representing the weight vectors of each subject's matrix on each of the obtained conceptual dimensions. Therefore, this procedure makes it possible to estimate whether different groups of subjects contribute differently to the different dimensions or whether the same subject contributes differently on different occasions. Hence, the weighted procedure was used to evaluate whether the mental representations of the Trace and Non-Trace groups differed, and if these representations changed as a function of programming experience.

Scaling solutions were obtained with two and three dimensions. A three-dimensional solution was selected because it yielded a reasonable correlation
between original and computed distances (0.68), a reasonable proportion of variance ($R^2 = 0.466$) and easy interpretability. Figures 1, 2 and 3 show the location of each concept in the space defined by two of the three dimensions (1–2, 1–3, 2–3). This space is common to all subjects and to both phases.

Inspection of these figures indicates that Dimensions 1 and 2 capture semantic differences. Dimension 1 clusters control statements (while, if-else, switch, case, break) in the lower quadrants, and input/output functions and data types in the upper quadrants of Figures 1 and 2. Therefore, this first dimension separates control statements from the rest of the concepts. Dimension 2 clusters input/output functions and control statements (putchar, getchar, while, if-else, etc.) in the right-hand quadrants of Figure 1 and in the lower quadrants of Figure 3, and data type (char, int, float, etc.) in the left-hand quadrants of Figure 1 and in the upper quadrants of Figure 3. Therefore, this second dimension separates data types from the rest of the concepts. These two dimensions are considered semantic because they cluster the concept based on the function they play in the programming language. Thus, as shown in Figure 1, both dimensions cluster the concept into three well-defined groups: (1) input/output functions, (2) data types, and (3) control statements. On the other hand, Dimension 3 seems to capture syntactic differences. Thus, this third dimension clusters the control statements “switch”, “case” and “break” that form a syntactic structure and separates them from the control statements “while” and “if-else”. Also, this dimension clusters each data type with its syntactically appropriate input/output function. For example, the data types
Figure 2. Locations of programming concepts in the space defined by Dimensions 1 and 3 in the solution of the rating task.

Figure 3. Locations of programming concepts in the space defined by Dimensions 2 and 3 in the solution of the rating task.
“char”, “array” and the input/output functions “putchar”, “getchar” are located on the left side of the dimension, whereas the data types “int”, “float” are located on the right side of the dimension.

Figures 4, 5 and 6 show subjects' weight vectors on each dimension and how the Trace (T) and Non-Trace (NT) subjects are distributed in the three-dimensional space on each phase. Although the figures show the end-points of the vectors, each point in those figures represents a weight vector for a particular subject (Trace or Non-Trace) in a particular phase (1 or 2).

Inspection of the figures indicates that the Trace and Non-Trace subjects are different in the way they are located along Dimensions 2 and 3. Thus, subjects in the Trace group tend to give greater weight to Dimension 2, whereas subjects in the Non-Trace group tend to give greater weight to Dimension 3. Figure 7 shows the mean weights of each group on each dimension. An angular analysis of variance with Trace–Non-Trace as a between-subjects variable was performed on the data (Shiffman, Reynolds & Young, 1981). The purpose of this analysis was to evaluate if the angular separation between the weight vectors was greater between groups than within groups. Two vectors that point in the same direction and, therefore, have a small angular distance would tell us that the two subjects do not differ in the importance they attach to the several dimensions. Two vectors with different orientations and, therefore, with relatively larger angular distances would correspond to two individuals who differ in dimensional salience. Hence, subjects
belonging to the same group should have smaller angular distances than subjects belonging to different groups.

With three-dimensional solutions, this angular analysis starts by locating all the data points in a sphere with radius = 1 and estimating the distance in degrees between each of the vectors. This angular distance is then entered into the analysis of variance. Results of this analysis indicated that there were significant differences between the Trace and Non-Trace groups \( F(2, 28) = 3.48 \), mean square error (M.S.E.) = 0.02. Since this analysis has not been developed for more than one independent variable, a second analysis with phase as a within-subjects variable was performed on the data. Results of this analysis indicated that there were no significant differences between phases \( F(2, 28) = 1.14 \), M.S.E. = 0.02.

Finally, as suggested by Schiffman, Reynolds and Young (1981), a multidimensional scaling analysis with the angular distances between each pair of subjects as the dependent variable was performed on the data. The angular distance between each pair of subjects was calculated as an index of subjects' similarity. A similarity matrix was then constructed with those distances and submitted to a multidimensional scaling analysis. According to Schiffman et al., if there are differences between the groups, a two-dimensional solution on the subjects' similarity matrix should separate subjects from both groups. Figure 8 shows the two-dimensional solution of this analysis (stress < 0.0001, \( R^2 = 1 \)).

With two exceptions (NT11 & NT12), the Non-Trace subjects are located left of
the zero point on Dimension 1, whereas most Trace subjects are located right of this point. Therefore, Dimension 1 seems to separate Trace from Non-Trace subjects, and indicates again that both groups seem to have different mental representations of the C programming concepts.

PROGRAMMING TASKS

Two judges jointly evaluated the programs and the verbal protocols offered by the subjects as they were writing the programs. Each subject received a score within each of the seven categories explained below. Judges discussed the performance of the subject on each category until they reached an agreement about what score to give.

1. Understanding of the program instructions. This category was intended to assess whether subjects' protocols showed a good understanding of what the program was supposed to do. For example, the program that the subjects had to write in the first phase had to discriminate data types (digit, character, etc.) and to count the number of each type introduced from the keyboard. Therefore, subjects received a point if they understood that the program should discriminate data types, and count the number of each data type entered.

2. Solution offered by the program. The score given to the subject within this
category was intended to reflect the goodness of the solution given by the subject to the problem posed by the program instructions. For example, judges scored on a scale from 1–5 the correctness of the solution (the higher the score, the more correct the solution) offered by the subjects to the discrimination and counting problems of the first program.

3. Input. This category was intended to evaluate whether the subjects used input functions, and if they did so correctly. Thus, subjects received a point if they used input functions, and, if they did, they received a score from 1–5 that reflected the correctness of their use.

4. Output. As in the previous category, subjects received a point if they used output functions and a score from 1–5 reflecting whether they did so in a correct way.

5. Variables. This category was intended to assess whether the selection of variable type was appropriate to the type of data (in which case subjects received a point) and to assess the correctness of the usage of variables within the program (scored from 1–5).

6. Control structures. The score within this category reflected whether subjects used any control statement (while, for, if-else, etc.) (in which case they received a point) and if the use of control structures in the program was correct (scored from 1–5).

7. Function. As in the previous category, it was assessed whether subjects created a function within the program (in which case they received a point) and whether they did so in a correct way (scored from 1–5).
Finally, the points obtained in each of the categories were summed for each subject, and this score was considered to reflect subjects' global performance on the program.

Table 1 shows the scores obtained by the Trace and Non-Trace groups within each category in each of the two phases of the experiment. These scores consist of the number of subjects from each group that obtained a point in that category or the mean score for the group within a given category or both. For example, the score in the understanding category consists of the number of subjects that understood the program; the score within the solution category consists of the mean judges' scores for the groups, and the scores for the remaining categories consist of both the mean scores and (in parentheses) the number of subjects that obtained a point within the category. A subject from the Trace group was eliminated from the analyses performed on the first program's score data because she did not understand what the program was supposed to do, and therefore, her performance was not equivalent to that of the rest of the subjects.

The bottom row shows subjects' mean total scores for each of the experimental phases. As shown, subjects' scores were lower in the first than in the second phase. Thus, subjects wrote better and more complete programs during the second learning phase. A Wilcoxon Signed Rank test for the effect of phases was significant ($W = 2.21, p = 0.01$). Also, there was a tendency for the Trace group to be better than the Non-Trace group in the first learning phase. The mean scores for the Trace
Table 1
Scores for the Trace and Non-Trace groups in the programming tasks

<table>
<thead>
<tr>
<th></th>
<th>First program</th>
<th>Second program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trace</td>
<td>Non-Trace</td>
</tr>
<tr>
<td>Understanding†</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Solution‡</td>
<td>2.12</td>
<td>2.00</td>
</tr>
<tr>
<td>Programming categories§</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>3.75 (4)</td>
<td>1.25 (4)</td>
</tr>
<tr>
<td>Output</td>
<td>3.75 (4)</td>
<td>2.25 (4)</td>
</tr>
<tr>
<td>Variables</td>
<td>1.50 (2)</td>
<td>0.00 (3)</td>
</tr>
<tr>
<td>Control structures</td>
<td>2.50 (2)</td>
<td>0.66 (3)</td>
</tr>
<tr>
<td>Function</td>
<td>2.33 (3)</td>
<td>1.00 (3)</td>
</tr>
<tr>
<td>Mean total scores</td>
<td>16.25</td>
<td>9.50</td>
</tr>
</tbody>
</table>

† Maximum of four subjects in each cell.
‡ Maximum of five points in each cell.
§ Maximum of five points in each cell, number of subjects that scored a point in the category in parentheses.

and Non-Trace groups during the first phase were 16.25 (S.D. = 11.32) and 9.50 (S.D. = 11.81), respectively. For the second phase, the mean scores were 27.25 (S.D. = 6.13) and 29.0 (S.D. = 6.97), respectively. However, these differences were not significant. Results of Mann–Whitney tests for the differences between the groups were U = 6.5, p > 0.5 for the first phase and U = 6.5, p > 0.5 for the second phase.

To further explore the possible differences between the Trace and Non-Trace groups, Mann–Whitney U-tests were performed on the scores within each of the categories in each of the learning phases. As can be seen, during the first phase, subjects in the Trace group showed a tendency to outperform subjects in the Non-Trace group in all the programming categories. However, the only significant difference between both groups was in the “Variables” category (U = 6.00, p = 0.05). The Mann–Whitney test results for the other categories were: U = 8.00, p > 0.10 for the “Solution” category; U = 10.50, p = 0.06 for the “Input” category; U = 9.00, p > 0.10 for the “Output” category; U = 4.00, p > 0.10 for the “Control structures” category; and U = 6.50, p > 0.10 for the “Function” category. In addition, during the second phase the slight differences in performance observed during the first phase disappeared. In most categories both groups obtained the maximum grade possible and only in the “Function” and “Solution” categories did the Trace group show a non-significant tendency to be better than the Non-Trace group. The Mann–Whitney test results for these categories were U = 10.50, p > 0.10 and U = 7.00 p > 0.10, respectively. Therefore, both Trace and Non-Trace subjects appeared to perform similarly, especially during the second phase.

Both groups improved their programming skills from one learning phase to the next, so as to perform very close to the maximum possible during the second phase. This high performance made it difficult to assess whether both groups had learned to the same degree or whether, given more difficult programs, differences would again become apparent.
Table 2

Results of the debugging task with number of subjects and mean time for finding the error and correcting it in the second phase of the experiment for the Trace and Non-Trace groups

<table>
<thead>
<tr>
<th>Presentation</th>
<th>On screen</th>
<th>On paper</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trace</td>
<td>Non-Trace</td>
</tr>
<tr>
<td><strong>Find the error</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of Subjects</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Time</td>
<td>2 min 29 s</td>
<td>2 min 38 s</td>
</tr>
<tr>
<td><strong>Correct the error</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of Subjects</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Time</td>
<td>2 min 1 s</td>
<td>1 min 48 s</td>
</tr>
</tbody>
</table>

Debugging Task

As in the programming task, two judges jointly scored subjects' protocols regarding whether they found the program error and whether they introduced changes in the program that corrected that error. The time to find and correct the error was also recorded. During the first phase both groups of subjects performed the task at floor level. Only one subject in the Non-Trace group found the error, after 46 min, and even upon finding it was not able to correct it. Table 2 shows the number of subjects who found and corrected the error and the mean time to find and correct it for each of the groups during the second phase in both paper and screen programs.

Obviously, both groups of subjects were much better at performing the task during this second phase. Seven of the eight subjects found and corrected the error when it was presented on the screen. And, although the subjects' performance was a little lower when the program was presented on paper, five of the eight subjects found and corrected it. The difference between the screen and paper condition did not reach significance ($\chi^2 = 0.69, p > 0.10$).

Both groups of subject seemed to be able to perform the task similarly. Mann–Whitney U-tests were performed for comparison between the groups in the time they needed to find and correct the programs. None of the between groups comparisons were significant. The Mann–Whitney test results for finding the error were $U = 6.00, p > 0.10$ in the screen condition and $U < 1.00, p = 0.08$ in the paper condition, and for correcting the error were $U = 6.00, p > 0.10$ in the screen condition and $U = 2.00, p > 0.10$ in the paper condition.

Discussion

The data suggest that the use of programming facilities such as the Trace, that make visible to the user the actions and operations taking place as the program runs, result in more semantically-oriented mental representations. In this study, subjects had different mental representations of the C programming concepts as a function of the group to which they belonged. Those subjects who used the trace facility during the
learning sessions (Trace group) seemed to acquire mental representations based more on semantic aspects of the programming language, whereas the subjects that did not use the trace facility (Non-Trace group) seemed to have mental representations based more on syntactic aspects of the language. The Trace and Non-Trace groups differed on the weights they attributed to Dimensions 2 and 3. As discussed, Dimension 2 separates type of data from control and input/output statements, and hence it captures semantic differences among the concepts. Dimension 3 separates control statements mainly based on syntactic differences among them, and therefore seems to capture syntactic differences among the concepts. Interestingly, the Trace subjects gave greater weight to the semantic dimension, whereas the Non-Trace subjects gave greater weight to the syntactic one. Hence, showing subjects the sequence of events as they occur in the program and the particular circumstances that lead the program to a given state seemed to increase subjects' understanding of the C concepts. The use of the trace facility allowed subjects to follow the flow of operations and actions of the program, and therefore to focus on the functions and way of working of each of the statements, variables and functions of the program. This better understanding became apparent in the semantic organization of the C concepts captured by the scaling solution.

This more semantically-based representation found for the Trace Group relative to the Non-Trace group has its parallel in the differences in the organization of programming knowledge found between experts and novices in other studies. In general, results have indicated that programming experts organize programming concepts based on their meaning, whereas novices organize the concepts according to syntactic or surface features (Adelson, 1981; McKeithen, Reitman, Reuter & Hirtle, 1981; Cooke & Schvaneveldt, 1989). Therefore, the mental representation of the Trace group more closely resembles the mental representation of experts than does the mental representation of the Non-Trace group. However, both of them had the same amount of training and programming experience.

The results might be of interest to those who teach programming. The learning phases of the present study were designed so that they closely resembled typical programming courses and have clear implications for training. As discussed, many claims have been made for the power of visualization techniques in facilitating learning, and several systems have been developed to show to the user the state of knowledge of the system (see Myers, 1990; Murray & McDaid, 1993, for good reviews). However, not too many studies have tested the efficacy of those systems in helping students to learn programming concepts. The present study suggests that a visualization technique such as the Trace helps students to acquire a more semantically-oriented mental representation of programming concepts. However, this representational advantage may not affect performance. In the present study, the Trace and Non-Trace groups acquired similar programming skills. In both programming and debugging tasks, the Trace and Non-Trace subjects showed similar levels of performance. Hence, the use of visualization systems (e.g. Trace) seems to have an impact on how students understand and organize concepts, but not on how they use them.

This pattern of results is interesting because it suggests a possible dissociation between the acquisition and understanding of conceptual-declarative knowledge and the use of this knowledge to perform a procedural task. Thus, although the Trace
and Non-Trace groups organized their knowledge of programming concepts according to different dimensions, both types of organization appeared to be equally efficient when performing procedural tasks (programming and debugging). Thus, the representational differences between the Trace and Non-Trace groups were not critical for performance.

This dissociation between mental representation and performance was also present when looking at the effect of phase. Both groups improved their programming skills from one learning phase to the next, so as to perform very close to the maximum possible during the second phase. Although this high performance makes it possible to attribute the lack of difference between the Trace and Non-Trace groups to ceiling effects, the different pattern found for the rating task suggests again a possible dissociation between mental representation and performance. Thus, although increased experience resulted in superior programming performance, there was no effect of experience on the subjects’ mental representation of the programming concepts. Thus, phase had an effect in the programming and debugging tasks, but did not change the scaling solution. Subjects wrote better programs after the second phase of the experiment than after the first one. Similarly, subjects were able to find a program error and correct it after the second phase, whereas many of the subjects were unable to do so after the first one. However, the conceptual organization reflected by the scaling solution to the relatedness tasks remained unchanged through the phases. This suggests different temporal courses in the acquisition of a mental organization of the programming concepts (mental model) and in the proper use of those concepts in programming.

The apparent independence of mental representation and performance found in the present study contrasts with other studies that have shown that knowledge is organized differently in memory as a function of expertise. However, those studies (Adelson, 1981; Cooke & Schvaneveldt, 1988) have compared groups of subjects that differed greatly in their programming skills, whereas in this study subjects’ mental representations were assessed twice for the same group of novice subjects. Although these subjects were acquiring some programming skill throughout the experiment, they were nonetheless novice programmers at the end of training. Therefore, it is possible that very early in learning, a mental model is acquired that remains constant until greater changes in programming skill have occurred. At these early stages, the type of mental model seems to be influenced by the learning conditions (e.g. Trace), but seems not to be a determinant of performance.

This research was supported by Grant DGICYT PB89-047 from the Spanish Government to the authors.

References


Paper accepted for publication by Associate Editor Professor N. Cooke.