Pre-publication draft of a paper which appeared in the journal of Educational Computing Research, Volume 22(2) pp 187-215

The Efficacy of a Low-Level Program Visualisation Tool for Teaching Programming Concepts to Novice C Programmers

Authors

Philip A. Smith
School of Information Technology and Mathematical Sciences
Mt Helen Campus
University of Ballarat
Ballarat, 3353
Victoria
Australia

E-mail – p.smith@ballarat.edu.au

Geoffrey I. Webb
School of Computing and Mathematics
Geelong Campus
Deakin University
Geelong, 3217
Victoria
Australia

E-mail - webb@deakin.edu.au

Keywords – System Evaluation, IT Education, Program Visualisation, Software Visualisation

The Efficacy of a Low-Level Program Visualisation Tool for Teaching Programming Concepts to Novice C Programmers

Abstract

It is widely agreed that learning to program is difficult. Program visualisation tools make visible aspects of program execution which are often hidden from the user. While several program visualisation tools aimed at novice programmers have been developed over the past decade there is little empirical evidence showing that novices actually benefit from their use [1]. In this paper we describe a "Glass-box Interpreter" called Bradman. An experiment is presented which tests the efficacy of Bradman in assisting novice programmers learn programming concepts. We show that students that used the glass-box interpreter achieved greater understanding of some programming concepts than those without access. We also give evidence that the student's ability to assimilate new concepts was enhanced by exposure to the glass-box interpreter. This is experimental confirmation that such tools are beneficial in helping novices learn programming.

Introduction

Program visualisation tools, which provide several views of a program and its execution, appear to be promising aids for teaching novice programmers. However, there is little empirical evidence regarding their efficacy [1]. This paper presents an experiment that provides such evidence. We show that students can benefit from access to such a tool both in achieving greater understanding of some programming concepts and also in their ability to assimilate new concepts.

Background

Many novice programmers experience difficulties and frustrations in their attempts to learn programming. Researchers in the field generally concede that programming is a difficult skill to master [2] [3]. Students experience difficulties developing, comprehending and debugging programs, often reaching impasses from which they cannot proceed without assistance.

Mayer [4] said that “meaningful learning” occurs when new knowledge is actively associated with appropriate pre-existing knowledge structures. If meaningful learning occurs then the learner will have “understood” the new knowledge. This process is called “assimilation”. If appropriate knowledge structures do not exist then the new knowledge needs to learned by “rote” which is memorisation without understanding. If knowledge is learned by rote then the learner is less likely to be able to apply the knowledge in new situations.

However, Dijkstra [5] described computers as a “radical novelty”. By this he meant that the computer represented such a “sharp discontinuity” in learning that analogies should not be drawn between it and more familiar concepts. He said that computing should be approached with a “blank mind” and gave several examples which he claimed gave evidence of the inappropriateness of conceptualising computing in familiar terms.

The above scenario is further complicated by the learners themselves. Each student brings into the learning experience a unique blend of knowledge, beliefs, fears and prejudices which will colour the way they learn the new material. Students will use their own metaphors to try to make sense of what they are learning [6]. They will also make inferences upon the instructor’s

analogies which were unintended by the instructor. For example, an instructor might use a box as an analogy to represent a memory location, and a student might make the inference that a memory location might store more than one value since a box can usually hold more than one item [7]. Pre-existing knowledge can even prevent students from seeing that a programming problem is a problem at all [8].

However, the provision of appropriate metaphors is a useful way to help students understand new material provided the students are made aware of the limitations of the metaphor. In this case metaphors are a form of conceptual model. Conceptual models are invented by educators to provide an anchoring framework upon which students can assimilate new knowledge [9]. The provision of conceptual models is an attempt to associate new learning material with more familiar concepts. Conceptual models are intended to make the student's mental model more useful when learning the target system. A mental model is an internal representation of the target system which provides predictive and explanatory power to the operator [9].

The educator will have several means at their disposal with which to teach the student by providing appropriate anchoring frameworks onto which the student can assimilate the new information. Some of these methods are:

- The instructor will give representations of various concepts on the blackboard. However, the blackboard is a static medium. This can cause difficulties when seeking to explain dynamic programming concepts. The blackboard "walkthrough" forces students to take messy notes from which it is difficult to repeat the walkthrough at a later stage [10].

- Students can use debugging tools in laboratory situations. Such tools are often commercial debuggers which are designed for experts and which, consequently, are often too complicated for novices. Certainly it is our experience that students use the UNIX symbolic debugger gdb with great reluctance.

- Students will also have access to written material such as text books and study guides. Again these are static media which are also unresponsive - if a student does not understand something the book cannot explain it in a different way.

- Human tutors, teaching on a one to one basis, are probably the best way to teach novices. Anderson and Reiser [2] reported that students with access to private tutors learned as much Lisp in eleven hours as other students did in forty three hours. However, access to human tutors in educational institutions is often very limited.

Another way to assist novice programmers is to provide computerised assistants which are created specifically for them. One class of such assistants are software visualisation tools. Software visualisation is the use of interactive computer graphics, typography, graphic design, animation, and cinematography to enhance the interface between computer programmers and their programs [11]. While a great deal of research has gone into the creation of these systems there has been little empirical data gathered regarding their efficacy [1]. One of the reasons for this is the difficulty in assembling the proper ingredients for such an evaluation [12]. Mulholland [1] attributes the fact that software visualisation systems are not widely used to lack of empirical evidence as to their value in actually teaching novice programmers.

Software visualisation tools can be sorted into two main groups:

- Algorithm animation [11] gives a graphical representation of the algorithm used to implement a program. Algorithm animations are to a large extent programming language independent. They are used to give students a visual representation of how an algorithm works. Each animation must be individually created.

- Program visualisation tools, on the other hand, are programming language dependent. They are used to animate low-level features of a program such as the source code and the changes of variable states. They should be able to automatically handle all possible programs that can be written in the target language.

The past few years have seen the emergence of several low-level program visualisation tools designed explicitly for novice programmers [2] [10] [13]. The systems were enthusiastically embraced by students but none of them have as yet have been tested under controlled conditions.

Bradman is a low-level program visualisation tool designed to provide a conceptual model of C program execution for novice programmers. It is an interpreter which makes visible aspects of the programming process which are normally hidden from the user. For this reason, we call it a “Glass-box Interpreter”. It is similar to “program animators” [14] except that Bradman is designed to enable students to develop their own programs as well as run those created by educators for pedagogical purposes. Bradman also incorporates novel features designed specifically to assist novice programmers. We now describe Bradman and then describe an experiment in which we explore Bradman’s effect on novice programmers. This experiment gives the first concrete

evidence that even a low-level program visualisation tool can assist a student's meaningful learning of C concepts.

Bradman

Bradman was developed as a tool to provide assistance for novice programmers in their endeavours to learn C. The School of Computing and Mathematics at Deakin University had recently switched from using Pascal as an introductory programming language to using C. It was felt that the problems that students were having with Pascal would be exacerbated when using C and that the tools available were not adequate for enhancing the student's mental representation of program execution. The motivation for developing Bradman was to produce an assistant that provides a useful conceptual model onto which students could assimilate new knowledge about programming.

Bradman provides a model which reinforces the view of the program achieving its results by the sequential change of program state caused by the execution of programming statements. This model is intended to assist students visualise the execution of programs more clearly thus enhancing their mental models of program execution.

Implementation

Bradman takes the user's syntactically correct source code as input. This source code is compiled producing an efficient internal representation in the form of a syntax tree. This internal representation is never presented to the user, the corresponding source code being referred to whenever appropriate. As each statement is executed code embedded in the run-time machine sends information to the various windows which provide

different views of the program. Four of these windows are displayed at all times while others appear when certain conditions arise. These windows are now described.

Code Window

The code window provides many of the features found by any state-of-the-art debugger. It displays the program code and shows the current point of execution by using a marker on the side. Vertical scrollbars enable the user to see different parts of longer programs. The window itself can be resized with the mouse if necessary.

One of the primary design goals that influenced the code window was to provide as much as possible of the functionality of a conventional visibility debugger with a minimum of complexity. Simplicity is sought throughout the system as we believe that novice programmers have tremendous cognitive loads imposed on them by the need to master the new computing paradigm and that it is important that the environments add the least possible amount to this load.

Variables Window

The code window makes explicit to the novice programmer the manner in which the execution of a statement affects the point of execution. In contrast, the variables window makes explicit the way in which the set of values of the variables is affected by a statement execution. While previous program visualisation tools have shown the values of the variables and how they change as execution proceeds, Bradman has added features which are intended to further reinforce the model of a program being an active entity

which achieves its results through the execution of program statements.

The set of variable/value bindings for the current program state can be altered by the execution of a program statement to produce a new set of values. This concept is conveyed by displaying the values of the variables in two columns - a before and an after column. The display shows the set of values before the execution of the statement and the set of values after the execution of the current statement. To reinforce the concept of this change of state being an ongoing process, it is explicitly shown that the after state of one statement is the before state of the next statement executed. Animation which shows the after column from one statement execution migrating across the display to become the before column for the next statement, is used to reinforce this concept. Finally, the values of the current set of before values are often (but not always) used to help create a new set of after values. This is shown by appropriate highlighting of the variables involved.

The variables window also differs from the variable display of a standard visibility debugger by including a formalism to better represent pointers. In general, the values of pointers are represented by large unsigned integers which denote memory locations. Standard visibility debuggers display these values. The program visualisation tool assists the novice’s understanding of the way pointers refer to other variables by an explicit display showing the connection between the two memory locations as is commonly done in blackboard presentations. VIPS [15] displays pointers in a similar manner.

In C, the programmer must initialise a variable before referencing its value. Failure to do so will usually result in an error. It is common for a student to assume that a variable's initial value is zero. Hence in the variable display, the variables which have yet to be assigned a value are explicitly marked as such. If a program attempts to reference an uninitialised variable then an error message is displayed.

Some students are confused by the use of functions in statements. The variables display explicitly shows how the function returns a value and how this value contributes to the value of the expression from which the function was invoked. Another feature of the variables window is that it explicitly shows the value of a function after it has returned to its calling statement. The function value appears in a box in a similar manner to a variable value and is shown to contribute to the value of an expression by the use of highlighting in a similar manner to that of variables.

Finally, the variables window explicitly represents the precedences of expressions. C is a language in which the precedences of operators is implicit, unless overridden by the use of parentheses, and not necessarily consistent. Some operators have left to right precedence, others right to left and others no precedence at all. Furthermore, some operators have different precedences to other operators. This can be confusing to novice programmers. Bradman uses parentheses to explicitly show the default precedences and hence the order in which sub-expressions are calculated.

Explanations window

The code window and the variables window provide a conceptual model for the programmer of which changes are effected in the

program state by the execution of a statement. However, they do not tell the user how the statement caused these changes. One can envisage situations in which a novice might see the changes wrought by a particular statement but still not understand why the statement had that effect. For example, consider the statement

```
x = 3 + 4 * 2 + 5;
```

The user might expect x to be assigned a value as follows:

```
x -> (3 + 4) * (2 + 5) -> 49
```

and be mystified when, in fact, the final value is 16,

```
x -> 3 + (4 * 2) + 5 -> 16
```

An explanation of how a statement achieves its results is used to reinforce the model. These explanations are specific to the context in which the actual statement is executed. Thus instead of explaining in general terms what an assignment statement does, the explanations provides information about what the current assignment statement is actually doing. For example, the statement

```
i = 10;
```

is explained not only as being an assignment statement but as an assignment statement in which the value 10 is assigned to the variable i. Birch et al. [14]in their description of the future directions for Dynalab, mentioned annotation which would provide a running commentary about the program being animated. Dynalab is a system in which the student mainly runs programs pre-written by the instructors. Hence, it is possible that they are designed at a higher level than that of Bradman’s explanations window. They also mentioned the possibility of using sound for this facility.

The explanations window gives contextualised information describing how each statement works. The information which is provided by the explanations window is embedded in the run-time machine which enables it to include context such as the values of variables and the memory locations to which pointers point. The information given by the explanations window is textual.

The explanations window uses a simple process by which information is added to it as the program is executed. It provides a lower level analysis than the other windows operating at the expression level rather than the statement level. This is necessary because its purpose is to explain the workings of individual statements. The cumulative explanation for the entire execution remains in the window, so that the user can scroll back to investigate the history of how they reached the current state. Hence the explanations window is also an execution trace. This trace is also saved to a file enabling the student to study it after the session with Bradman has terminated.

Input/Output Window

This window provides a mechanism by which the user can communicate necessary input and output to the program. While this is mainly straightforward, this window makes visible two aspects of input that are usually hidden from the user.

First, when a statement requiring input from standard input is executed and the input buffer is empty the program will wait until input is entered. This can confuse novices if they have not coded an appropriate prompt for input into their program, causing them to think that there is something wrong with their program. While a program is waiting for standard input to be entered a flashing

message will appear at the bottom of the input/output window telling the user to enter input. The message

Program requires input

flashes on and off prompting the user to enter input. This message is designed to explicitly remind the user that input is required, so that the user does not mistakenly assume that the program has crashed or suspended. It will remain in effect until the user has entered enough input to give values to all three input variables. Second, when a user running the program normally enters more standard input than the program is ready to utilise, the excess input is stored in a buffer and used if more input statements requiring standard input are executed.

This buffer is normally invisible and novice programmers can become confused when input variables are assigned values that they did not intend for them. To clarify this for the novice, buffered input is displayed at the bottom of the input/output window.

Error Window

The error window appears only when a program independent error has been detected in the execution. For example if there is an attempt to reference a variable before it has been assigned a value then the error window will appear describing the error and suggesting ways it might be fixed.

The error window will only appear if Bradman attempts to execute the faulty statement so an error might be missed if execution does not happen to encounter it. Once an error is reported it is necessary for the user to correct it and then restart the program. Errors that are reported include:

- An attempt to reference variables that have not been assigned a value.
- An attempt to use an incompatible format in an input/output statement.
- A discrepancy between the number of format characters in an input/output statement and the number of arguments.
- An attempt to divide by zero.
- An attempt to assign an expression to a variable with a different level of indirection.

Bradman will report an uninitialised variable on the first attempt of the program to reference such a variable. In many debuggers such an error will only be reported if the failure by the programmer to explicitly assign a value to c causes an attempt to divide by zero or similar error. In this case a more useful error message is provided by Bradman.

Edit Window

The edit window, invoked from the code window, is a facility which allows the user to modify the program within the Bradman environment. It is simply a window which initiates a session in an external editor with the user's source code. The user can modify the program and quit from the window. Once the user quits from the window a new session is started with the modified source code now appearing in the code window. In its current version, Bradman will finish the session and exit if the user's modified program contains a syntax error.

Experiment

This experiment attempts to evaluate whether access to a glass-box interpreter assists the user to develop a better understanding of program execution, in other words, whether it provides an adequate knowledge framework of program execution onto which students can assimilate the new information. Access to software visualisation tools is often claimed to be of benefit to novice programmers but little empirical evidence of their value is provided [1]. This is true of both program visualisation tools and algorithm animators.

Methodology

Volunteers were sought from Deakin University's introductory programming course which teaches programming concepts using C. The experiment ran over a three week period in first semester. Subjects were required to attend three two-hour laboratory sessions (one per week). They were told that they would be testing ways to improve programming environments for novice programmers. A payment of thirty Australian dollars was made to the participants who completed all three sessions. Twenty-six people volunteered to participate. However, two subjects from the control group withdrew after week two, leaving unequal groups - one of thirteen and another of eleven. All subjects completed an appropriate consent form before commencing the experiment

The laboratory sessions were based on those already prepared by the instructors for the introductory unit [16]. These sessions required the students to perform desk-checking of pre-written programs. This format was chosen for the experiment for the following reasons:

- Many researchers report that the ability to desk-check programs is an important difference between novice and expert programmers [9].

- Laboratory sessions involving desk-checking were already part of the introductory programming unit at Deakin University. It was straightforward to modify the format of these laboratory sessions for the purposes of the current research. Volunteers were put at minimum inconvenience because they were able to substitute the experimental laboratory sessions for their normal sessions.

- It allows simple extraction of quantifiable data.

- Students must have some understanding of certain programming concepts before they can successfully desk-check programs involving these concepts. Thus it is possible to test, albeit indirectly, the student's understanding of these concepts as well as their ability to desk-check programs.

The first session was an introduction to the format that the subsequent sessions would take. All experimental data were collected in the second and third sessions. The authors of the unit intended that the students do the following:

- Desk-check a program prepared for them that was designed to illustrate and reinforce a concept which was introduced to them in lectures.

- Attempt to calculate the outputs of this program

- When finished, compile and run the program and compare the output to their calculations.

- Attempt to discover the reason for any mistakes that they made. They were allowed to use textbooks, ask questions of the tutors and analyse the program using the symbolic debugger, gdb. The students were given some time to analyse the program code. Then the instructors gave a demonstration on the whiteboard of how the program achieved its results.

This format was modified slightly for the experiment in the following ways:

- While the students were desk-checking the program they were required to answer multiple choice questions regarding the outputs.

- When they finished working on their program and had watched a demonstration on the whiteboard they were given a similar but different program to desk-check and with regard to which to answer multiple choice questions.

Thus the experimental period involved four tests - one at the beginning and one at the end of both sessions 2 and 3. For convenience sake these tests were numbered 1, 2, 3 and 4 in the order in which they were performed. Hence Test1 was conducted at the beginning of session 2 before the participants had received either treatment. Test4 was conducted at the end of session 3 after all experimental interventions had been performed and serves as a post-test. Test2, conducted immediately after the first intervention and test3 conducted a week later immediately before the second intervention allow us to map student progress through the experiment.

The students' proficiency was judged by the number of correct responses to the multiple choice questions. The twenty six

volunteers were split into two groups of thirteen. This was done based on the order in which they volunteered to participate - the first volunteer was put into the test group, the second into the control group, the third into the test group and so on. However, two subjects from the control group withdrew after week two, leaving unequal groups - one of thirteen and another of eleven. Experimental intervention related to the normal laboratory activities which were conducted between the tests conducted at the beginning and end of each session. The test group (consisting of thirteen people) had access to Bradman during this phase in which they attempted to gain an understanding of the program. The control group (consisting of eleven people) did not have access to Bradman during this phase.

The test group had access to a modified version of Bradman in which the explanations window was not available. An experiment which tested the efficacy of the explanations window was reported previously [17]. The explanations window was not available in the current experiment because we wished to focus on the value of the glass-box interpreter without confound factors, relating to the addition of textual elements.

Session 1

Session 1 was an introductory session in which the students were introduced to the format that would be used in sessions 2 and 3. The students (from both test and control groups) were required to desk-check a very simple program and answer multiple choice questions regarding its output. During the treatment period the test group was instructed on the use of Bradman - how it was invoked and how it could be used. During this time they also learned the material for the session. The control group learned

the material only during this period. At the end of the session all students, from both groups, were required to desk-check another program and answer multiple choice questions. The two test periods were intended solely to familiarise the students with the format of the following two sessions. The results were not collated or analysed. The control group underwent the same process as the test group except that they were not given exposure to Bradman. Although this first session might be seen as having an influence on the overall result, the effect is probably minimal because the students were given simple exercises. The comparative performance of the two groups through the experiment seems to support this. Even if this is not accepted, and the exposure to the low-level program visualisation tool in session 1 is viewed as a significant experimental intervention, while this devalues test1 as a pre-test it in no way devalues the final result which should then be viewed as the result of three rather than two experimental evaluations.

Session 2 - The Scope of Variables

The scope of variables is a concept that students often have trouble understanding. Students have to understand pieces of code in which a name refers to a global object in one statement while in another statement the same name refers to a different, local, object. They need to understand that a local variable in a calling function cannot be accessed (except through the use of pointers with which they were not as yet familiar) during the life of the called function. Hence desk-checking such a program is not a trivial exercise and presents difficult challenges to the novice programmer.

The program used in session 2 was prepared by the authors of the introductory programming unit as part of their course. It was incorporated in the experiment, with their permission, and was used as Test1. This program was designed to illustrate many of the concepts of scoping including how different variables can have the same name, how variables have a certain “life” or scope during which they are valid and how they lose their validity outside of this scope.

The program was developed by professional programming instructors to teach novice programmers about scoping. It consisted of four functions including the main function. The functions did not perform a meaningful task. They simply assigned and reassigned values to variables outputting them at various stages. None of the functions or variables had meaningful names. While the students were desk-checking this program they were required to answer five multiple choice questions. The students were instructed to circle the alternative that best answered the question. These questions along with the code for this program can be found in the Appendix.

An additional program was developed specifically for the study and used as Test2. It was intended to be similar but different to Test1. Again the students were asked to desk-check the program and again answer five multiple choice questions regarding its output.

Intervening Period Between Sessions 2 and 3

There was an interval of one week between the performance of Test2 and Test3. This break was significant because it gave an opportunity to judge whether Bradman indeed provided the student an adequate framework onto which to attach new information. During this period the students attended lectures in which the concepts used in session 3 were taught. If the glass-box interpreter was

efficacious in enhancing a student's mental model of program execution then one would expect the test group to perform better than the control group on Test2. One would also expect the test group to show an improvement on the initial test program in session 3.

Session 3 - Parameter Passing

In session 3 the subjects were called upon to analyse programs which involved pass-by-reference parameter passing. The instructors of the introductory programming unit, thought this concept to be of such difficulty and importance that they gave the students the following warning:

"The concept of 'pass-by-reference' (pointers) is probably the biggest hurdle you'll come across in C programming. When you absorb and UNDERSTAND this topic, you will have overcome the biggest learning curve in C. The rest of C programming will seem somewhat easier." [16]

These sentiments have been echoed by other researchers [18]. Pass-by-reference parameter passing in C involves the use of pointers which were a relatively new concept for the students being tested. They needed to develop a model of how, through the use of the pointers, the values of variables in the calling function would change rather than those in the called function.

Test3 consisted of two programs prepared by the instructors of the introductory unit, for use in the regular laboratory session. The first program consisted of an incorrect version of a swap program in which the parameters passed to the function swap were pass-by-value parameters. The second is the correct version in which pass-by-reference parameter passing is used. Test4 comprised three

additional programs developed specifically for the study. They differed from the Test3 programs only in the arguments that were passed to the function swap. The students were asked to answer two questions at the end of each program. The questions were the same for all five programs. These questions revolved around the values of the parameters after the swap had been completed but before the function returned and the values of the arguments of the call to swap after the function had returned. Unlike the programs in session 2, the function and variable names were more meaningful, reflecting their tasks.

Results and analysis

The multiple choice questions were collected and marked. The number of correct and incorrect responses for both groups are summarised in Tables 1 and 2. It must be remembered that the students had a selection of four possible responses from which to choose. All of the incorrect responses were grouped into one tally. Hence, for every test the distributions represent better than random performance by the participants.

The results seem to show an improvement on the part of the test group. As expected the number of correct responses for both groups increased after the intervention but the improvement of the test group was greater. The test group performed less well on Test1 (equal correct but more incorrect) than the control group but performed better on the Test2 (more correct and fewer incorrect).

However, it could be misleading to compare the groups in this manner because each individual was required to answer five questions. Thus a change in performance by just one subject could affect as many as five question responses. In order to better compare the two groups, the individuals were given a ranking

according to the number of correct response they made. Rankings were made for both Test1 and Test2.

Test1				
	Test Group		Control Group	
	Correct	Incorrect	Correct	Incorrect
Q1	4	9	5	6
Q2	7	6	4	7
Q3	6	7	8	3
Q4	4	9	4	7
Q5	5	8	5	6
Totals	26	39	26	29
Test 2				
	Test Group		Control Group	
	Correct	Incorrect	Correct	Incorrect
Q1	13	0	9	2
Q2	11	2	8	3
Q3	9	4	7	4
Q4	9	4	7	4
Q5	5	8	3	8
Totals	47	18	34	21

Table 1: Summary of Session 2 Results

Mann-Whitney U-tests comparing the two groups were performed on both of these rankings. The results were as follows:

Test1 $z = 0.70$ $p = 0.25$

Test2 $z = 0.96$ $p = 0.17$

Hence, although the figures from Table 1 appear to show an improvement in favour of the test group, statistical analysis of the rankings did not show that a situation was reached in which the test group performed significantly better than the control group.

The results for session 3 were summarised in Table 2. Again the individuals were given a ranking according to how many correct responses they made.

Rankings were made for both Test3 and the Test4 results and Mann-Whitney U-tests were performed on both of the rankings. The results of these tests were as follows:

Test3 $z = 1.42$ $p = 0.08$

Test4 $z = 2.03$ $p = 0.02$

The figures show a definite bias towards the test group. In Test1 the test group performed slightly worse than the control group. However, in Test3 the test group performed much better although significance cannot be claimed. (It should be noted that the power of these tests is low due to the small number of subjects and hence, there is a sizeable chance that an underlying advantage should fail to have been reflected in a significant value for p). In Test4 the test group performed significantly better (at the 0.05 level) than the control group.

Despite the lack of statistically significant outcomes in tests 2 and 3, it is tempting to hypothesise about the causes for the enhanced performance of the test group in Test3, (if one assumes that it does reflect an underlying advantage to the test group). In the period between session 2 and session 3 the students attended a lecture explaining the relevant concepts, especially

pass-by-reference parameters. Since the only difference in the treatment of the two groups was the test group's access to Bradman it is reasonable to assume that the glass-box interpreter prepared them in some way to better understand the information presented in the lecture.

Test 3					
		Test Group		Control Group	
		Correct	Incorrect	Correct	Incorrect
Program 1	**Q1**	7	6	6	5
	Q2	7	6	4	7
Program 2	**Q1**	8	5	3	8
	Q2	10	3	6	5
	Totals	32	20	19	25
Test 4					
		Test Group		Control Group	
		Correct	Incorrect	Correct	Incorrect
Program 3	**Q1**	11	2	7	4
	Q2	8	5	3	8
Program 4	**Q1**	10	3	7	4
	Q2	10	3	4	7
Program 5	**Q1**	10	3	5	6
	Q2	9	4	7	4
	Totals	58	20	33	33

Table 2: Summary of Session 3 Results

Mayer [4] has said that learners require a pre-existing framework onto which to attach the new knowledge. It is plausible to suggest that the use of a program visualisation tool facilitated the development of a cognitive framework that aided assimilation of

the new knowledge. This suggests that the tool provides an appropriate conceptual model of program execution. Irrespective of whether one places any weight on the apparent advantage for the test group in test3, test 4 reveals clearly that after the second experimental intervention (laboratory 3) the test group did enjoy a significant advantage. This is clear evidence that Bradman's form of low-level program visualisation actually influenced the students' mental model in a positive way.

As mentioned earlier, two students withdrew from the experiment after session 2 was conducted. As a result, their results could not be used and the groups became unequal with thirteen in the test group and eleven in the control group. However, there is no reason to believe that this would have confounded the results. The results show that in Test1 the control group, without the two subjects who withdrew, performed slightly better than the test group although this difference was not significant. Subsequent results show a clear improvement of the test group's ability to perform the given tasks as compared to that of the control group. It is not clear how these results could be interpreted as indicative of a confound introduced through the subjects' withdrawal.

Results of survey

The students' reaction to Bradman was very positive. This mirrors similar student enthusiasm for other program animators such as Dynalab [14]. The subjects who had access to Bradman were asked to complete a survey form (after the second and third sessions) to test their reactions and to enable them to provide comments. The survey consisted of two parts.

For the first part, subjects were required to give one of five responses to the following statements.

1 - Bradman is easy to use.

2 - Use of Bradman increased my general understanding of computer programming.

3 - I now find it easier to visualise how a program actually works

4 - Bradman would be of assistance for students developing programs (eg. for assignments)

5 - Use of Bradman in lectures would help students understand programming.

For each of the above statements the subject was required to give one of the following responses:

- Strongly agree
- Agree
- Neutral
- Disagree

Strongly disagree The responses are summarised in Table 3. As can be seen there was no negative response on any of the five questions. Only one student was neutral about whether his/her general understanding of programming had increased through the use of Bradman. The students all strongly agreed that they would find the system useful in developing their programs for their assignments. The results for session 3 were almost exactly the same. The only difference was that three of them were slightly

less positive that Bradman would help them with their assignments although they all agreed that they thought it would.

Results for Session 2					
Question	**Strongly Disagree**	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
1	0	0	0	6	7
2	0	0	1	5	7
3	0	0	0	3	10
4	0	0	0	0	13
5	0	0	0	4	9
Totals	0	0	1	18	46
Results for Session 3					
Question	**Strongly Disagree**	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
1	0	0	0	6	7
2	0	0	1	5	6
3	0	0	0	3	10
4	0	0	0	3	10
5	0	0	0	4	9
Totals	0	0	1	21	43

Table 3: Results of Survey for Sessions 2 and 3

The second part of the survey enabled the students to make free form comments regarding their experience with Bradman. This section consisted of three questions

- What features (if any) of Bradman did you find particularly useful?
- What features (if any) would you add to Bradman to make it more useful?
- What features (if any) of Bradman did you not like?

Most of the comments were in response to the first question. There was almost unanimous approval for the visibility of the variable states. Many of the students also explicitly mentioned that they liked the format showing the before and after columns. There were several comments giving approval for the "graphical representation" of the program.

In response to the second question the response, given by several people, was that they would like to be able to "go back to previous statements". Reverse execution is a feature that was considered but rejected during the design of Bradman. However, that the novice programmers themselves believe it would be of benefit strengthens the case for future evaluation of such a feature.

The main response to the third question was that they did not like the fact that Bradman was not universally available. This criticism was interpreted as a positive reflection on Bradman per se.

Other comments were of a general nature and were invariably positive. One of the students said "it helped me understand programs that I could not normally understand". Another said in response to the third question, "None! It's too good."

General Observation and Discussion

Bradman was enthusiastically accepted by the students, many of whom asked for it to be made generally available. More than one Bradman user told the experimenter that they had no idea about how programs worked until they used Bradman. This response is something that we expected, however, from people who were perhaps unsure about how well they were doing in their course. We expected them to appreciate the visibility and simplicity of Bradman after struggling with tools like gdb which are designed for expert programmers.

One interesting observation occurred during session 3, which was a session in which an assignment was due. Several of the students used Bradman, not to develop their assignments, but simply to watch their own programs execute. There was no need for them to do this because they were not looking for errors - they knew that their programs gave correct results. This agrees with an observation of Ross [10] who noted similar behaviour even in expert programmers and engineers. People seem to enjoy watching how their creations work.

The response was almost universal that they liked watching the way that the variables changed value as the program executed. They did not mention the other part of the program state change - the change of execution point. One student said that the marker in the code window should be changed to highlighting because it was too difficult to see.

One interesting aspect of the results was the short amount of time it took for the students to show signs of improvement. Other studies (for example, [19]) cover periods of a term or more. Jones [6] said that novice programmers had mental models that were

highly unstable and that changed rapidly. It is plausible to conclude that Bradman supplied the students with a more stable framework onto which to base their understanding of C hence speeding their development.

Future Directions

The experiment described in this paper provides evidence that novice programmers can benefit from the use of a glass-box interpreter. However, the experiment did not show which features of Bradman made the greatest contributions toward this effect. A major focus of our future work will be to test individual features of low-level program visualisation tools. We have previously provided subjective evidence [17] that students believe that they benefit from access to the explanations window. We want to obtain performance measures which demonstrate the utility of the explanations window and the variables window in particular.

We are also interested in seeing with which program skills a glass-box interpreter provides benefit. The experiment we conducted provided evidence that students benefit from exposure to Bradman when performing tasks which require desk-checking skills. However, learning to program involves the development of many skills including coding and debugging. We wish to conduct experiments which explore the efficacy of a glass-box interpreter in the development of these skills.

Conclusions

Programming is a difficult skill to master and novices need to develop appropriate knowledge structures to enable them to cope with it. Software visualisation tools are a class of computerised tool which show promise in assisting novices develop appropriate

models of programming. Program visualisation tools have been used to teach students in classroom and laboratory situations. However, there is little empirical evidence as to their efficacy and as a result their use is not widespread.

The experiment that we conducted provides evidence that low-level program visualisation tools, such as glass-box interpreters, can be beneficial in teaching novice programmers. This experiment gave concrete empirical evidence that such a tool can provide assistance in learning new programming concepts. It also indicated that the use of such a tool enables students to assimilate new information more effectively. This suggests that our tool presents a conceptual model which provides an appropriate framework onto which learners can assimilate new information. Bradman provides information in a dynamic manner while being very simple to use and makes visible aspects of program execution that are normally hidden. We believe that these features are important in enabling students to better visualise how a program works as it executes. We further believe that the benefits shown in comprehension will be mirrored in benefits to program development and program debugging and this will be the basis of further research.

References

[1] MULHOLLAND, P. A. (1995). A framework for describing and evaluating software visualisation systems: A case study in PROLOG. *Phd Thesis, Knowledge Media Institute*, Open University.

[2] ANDERSON, J. R. AND REISER, B. J. (1985). The Lisp Tutor. *Byte*, April, 159-175.

[3] BONAR, J. AND SOLOWAY, E. (1989). Uncovering Principles of Novice Programming. *Communications of the ACM*, 10-13.

[4] MAYER, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *Computing Surveys*, **13(1)**, 121-141.

[5] DIJKSTRA, E. (1989). On The Cruelty Of Really Teaching Computing Science. *Communications of the ACM*, **32(12)**, December, 1398-1414.

[6] JONES, A. (1982). Mental Models of a first programming language. *CAL Research Group Technical Report No. 29*, The Open University.

[7] PUTNAM, R. T., SLEEMAN, D., BAXTER, J. A. AND KUSPA, L. (1989). A Summary of Misconceptions of High School Basic Programmers. In E. Soloway and J. C. Spohrer, Eds. *Studying the Novice Programmer*, Hillsdale, New Jersey: Erlbaum. 301-314.

[8] EISENSTADT, M AND BREUKER, J. (1992). Naive Iteration: An account of the Conceptualisations Underlying Buggy Looping Programs. In M. Eisenstadt, M. T. Keane and T. Rajan Eds. *Novice Programming Environments: Explorations in Human Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.

[9] NORMAN, D. A. (1983). Some Observations on Mental Models. In D. Gentner and A. L. Stevens, Eds., *Mental Models*, Hillsdale, New Jersey: Erlbaum., 7-14.

[10] ROSS, R. J. Experience with the DYNAMOD Program Animator. (1991). *Special Interest Group on Computer Science Education*, **23(1)**, 35-42.

[11] PRICE, B. A., SMALL, I. S. AND BAECKER, R. M. (1983). A Principled Taxonomy of Software Visualisation. *Journal of Visual Languages and Computing*, **4(3)**, 211-266.

[12] STASKO, J., BADRE, A. AND LEWIS, C. (1993). Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. In *Proceedings of INTERCHI '93*, 61-66.

[13] FREUND, S. N. AND ROBERTS, E. S. (1996). THETIS: An ANSI C Programming Environment Designed for Introductory Use. *Special Interest Group on Computer Science Education*, **28(1)**, 300-304.

[14] BIRCH, M. R., BORONI, C. M., GOOSEY, F. W., PATTON, S. D., POOLE, D. K., PRATT, C. M. AND ROSS, R. J. (1995). DYNALAB, A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation. *Special Interest Group on Computer Science Education*, **27(1)**, 29-33.

[15] ISODA, S., SHIMOMURA, T. AND ONO, Y. (1987). VIPS: A Visual Debugger, *IEEE Software,* 8-18.

[16] DEW, R. A. AND NEWLANDS, D. A. (1996). Basic Programming Concepts. *Deakin University study guide*.

[17] SMITH, P. A. AND WEBB, G. I. (1995). Transparency Debugging With Explanations for Novice Programmers. *Proceedings of the 2nd Workshop on Automated and Algorithmic Debugging*, St.Malo, France.

[18] FLEURY, A. (1991). Parameter Passing: The Rules the Students Construct. *Communications of the ACM*, 283-286.

[19] CANAS, J. J., BAJO, M. T. AND GONZALVO, P. (1994). Mental Models and computer programming. *International Journal of Human-Computer studies*, **40**, 795-811.

Appendix

Session 2

program 1

```
int function1 (int);
int function2 (int);
int function3 (int);

int a, b;

int function1 (int a) {
       int b;

       b = 3 * a;
       printf (“\nfunction1: a = %d\n”, a);
       printf (“function1: b = %d\n”, b);
       return (function2 (b));
}

int function2 (int d){
       int c;

       c = 4;
       a = c + d;
       {
            int a;

            a = b + d;
            printf (“\nfunction2a: a = %d\n”, a);
            printf (“function2a: b = %d\n”, b);
            printf (“function2a: c = %d\n”, c);
            printf (“function2a: d = %d\n”, d);
       }
       printf (“\nfunction2b: a = %d\n”, a);
       printf (“function2b: b = %d\n”, b);
       printf (“function2b: c = %d\n”, c);
       printf (“function2b: d = %d\n”, d);
       return (function3 (a));
}

int function3 (int a){
       int d;

       d = a - 2;
       printf (“\nfunction3: a = %d\n”, a);
       printf (“function3: b = %d\n”, b);
       printf (“function3: d = %d\n”, d);
       return (d);
}

void main (void){
       int d;

       a = 2;
       b = 7;
```

```
        d = 10;

        printf (“main_a: a = %d\n”, a);
        printf (“main_a: b = %d\n”, b);
        printf (“main_a: d = %d\n”, d);

        d = function1 (d - b);

        printf (“\nmain_b: a = %d\n”, a);
        printf (“main_b: b = %d\n”, b);
        printf (“main_b: d = %d\n”, d);
}
```

1/ In the function function1 the output of the two printf statements (lines 2 and 3) is

a/ 3 6
b/ 3 9
c/ 2 6
d/ 2 7

2/ In the function function2 the output of the four printf statements (lines 10, 11, 12 and 13) is

a/ 16 9 4 10
b/ 16 7 4 9
c/ 13 7 4 9
d/ 13 9 4 9

3/ In the function function3 the output of the three printf statements (lines 20, 21 and 22) is
a/ 16 7 11
b/ 16 9 11
c/ 13 7 9
d/ 13 7 11

4/ In the function function2 the output of the four printf statements (lines 14, 15, 16 and 17) is
a/ 13 7 4 9
b/ 16 9 4 9
c/ 16 7 4 9
d/ None of the above

5/ In the main function the output of the three printf statements (lines 31, 32 and 33) is
a/ 2 7 10
b/ 13 9 11
c/ 13 7 11
c/ 2 9 10

program 2

```
#include <stdio.h>

int function1 (int);
int function2 (int);
int function3 (int);

int a, b;

int function1 (int a) {
       int b;

       a = 10;
       b = 3 * a;
       printf (“\nfunction1: a = %d\n”, a);
       printf (“function1: b = %d\n”, b);
       return (function2 (a));
}
int function2 (int d){
       int c;

       c = 4;
       a = c * d;
      {
            int a;

            printf (“\nfunction2a: a = %d\n”, a);
            printf (“function2a: b = %d\n”, b);
            printf (“function2a: c = %d\n”, c);
            printf (“function2a: d = %d\n”, d);
       }
       printf (“\nfunction2b: a = %d\n”, a);
       printf (“function2b: b = %d\n”, b);
       printf (“function2b: c = %d\n”, c);
       printf (“function2b: d = %d\n”, d);
       return (function3 (c));
}
int function3 (int a){
      int d;

       d = a * 2;
       printf (“\nfunction3: a = %d\n”, a);
      printf (“function3: b = %d\n”, b);
      printf (“function3: d = %d\n”, d);
      return (d + a);
}
void main (void) {
       int d;

       a = 11;
       b = 2;
       d = 11;

       printf (“main_a: a = %d\n”, a);
       printf (“main_a: b = %d\n”, b);
       printf (“main_a: d = %d\n”, d);

       d = function1 (a + d);
```

```
        printf ("\nmain_b: a = %d\n", a);
        printf ("main_b: b = %d\n", b);
        printf ("main_b: d = %d\n", d);

}
```

1/ In the function function1 the output of the two printf statements (lines 3 and 4) is

a/ 3 9 b/ 11 33
c/ 10 30 d/ 22 66

2/ In the function function2 the output of the four printf statements (lines 13, 14, 15 and 16) is

a/ 32 2 4 30
b/ 11 30 4 10
c/ 40 2 4 10
d/ 120 2 4 30

3/ In the function function3 the output of the three printf statements (lines 19, 20 and 21) is

a/ 11 2 8
b/ 40 2 10
c/ 11 30 10
d/ 4 2 8

4/ In the function function2 the output of the four printf statements (9, 10. 11 and 12) is

a/ 30 7 4 10
b/ 22 30 4 10
c/ 12 20 4 10
d/ none of the above

5/ In the function main the output of the three printf statements (lines 30, 31 and 32) is

a/ 40 2 12
b/ 2 11 11
c/ 32 2 4
d/ 11 30 8

Session 3

program 1

```
void main (void)
{
      void swap ();

      int a = 3, b = 6;

       printf (“main: a = %d, b = %d\n”, a, b);
       swap (a, b);
       printf (“main: a = %d, b = %d\n”, a, b);
}

void swap (int a, int b)
{
       int temp;

       temp = a;
       a = b;
       b = temp;

      printf (“swap: a = %d, b = %d\n”, a, b);
}
1/ The output of the printf statement at line 8 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
2/ The output of the printf statement at line 4 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
```

program 2

```
void main (void)
{

      void swap ();

      int a = 3, b = 6;

      printf (“main: a = %d, b = %d\n”, a, b);
      swap (&a, &b);
      printf (“main: a = %d, b = %d\n”, a, b);
}
void swap (int *a, int *b)
{
       int temp;

      temp = *a;
      *a = *b;
      *b = temp;

       printf (“swap: a = %d, b = %d\n”, *a, *b);
}
1/ The output of the printf statement at line 8 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
2/ The output of the printf statement at line 4 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
```

program 3

```
#include <stdio.h>
void main (void)
{
      void swap ();

      int a = 3, b = 6;

      printf (“main: a = %d, b = %d\n”, a, b);
      swap (a, &b);
      printf (“main: a = %d, b = %d\n”, a, b);
}
void swap (int a, int *b)
{
      int temp;

      temp = a;
      a = *b;
      *b = temp;

      printf (“swap: a = %d, *b = %d\n”, a, *b);
}
1/ The output of the printf statement at line 8 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
2/ The output of the printf statement at line 4 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
```

program 4

```
#include <stdio.h>
void main (void)
{
      void swap ();

      int a = 3, b = 6;

      printf (“main: a = %d, b = %d\n”, a, b);
      swap (&a, b);
      printf (“main: a = %d, b = %d\n”, a, b);
}
void swap (int *a, int b)
{
      int temp;

      temp = *a;
      *a = b;
      b = temp;

      printf (“swap: a = %d, b = %d\n”, *a, b);
}
1/ The output of the printf statement at line 8 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
2/ The output of the printf statement at line 4 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
```

program 5

```
#include <stdio.h>
void main (void)
{
      void swap ();

       int a = 3, b = 6;

       printf (“main: a = %d, b = %d\n”, a, b);
       swap (&b, b);
       printf (“main: a = %d, b = %d\n”, a, b);
}
void swap (int *a, int b)
{
       int temp;

       temp = *a;
       *a = b;
       b = temp;

       printf (“swap: a = %d, b = %d\n”, *a, b);
}
1/ The output of the printf statement at line 8 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
2/ The output of the printf statement at line 4 is
a/    swap: a = 3       b = 6
b/    swap  a = 6       b = 3
c/    swap  a = 6       b = 6
d/    swap  a = 3       b = 3
```