A Software Tool to Visualize Verbal Protocols to Enhance Strategic and Metacognitive Abilities in Basic Programming

doi:10.3991/ijim.v5i3.1668

Carlos Argelio Arévalo Mercado, Estela Lizbeth Muñoz Andrade and Juan Manuel Gómez Reynoso
Universidad Autónoma de Aguascalientes, Aguascalientes, México

***Abstract*—Learning to program is difficult for many first year undergraduate students. Instructional strategies of traditional programming courses tend to focus on syntactic issues and assigning practice exercises using the presentation-examples-practice formula and by showing the verbal and visual explanation of a teacher during the "step by step" process of writing a computer program. Cognitive literature regarding the mental processes involved in programming suggests that the explicit teaching of certain aspects such as mental models, strategic knowledge and metacognitive abilities, are critical issues of how to write and assemble the pieces of a computer program. Verbal protocols are often used in software engineering as a technique to record the short term cognitive process of a user or expert in evaluation or problem solving scenarios. We argue that verbal protocols can be used as a mechanism to explicitly show the strategic and metacognitive process of an instructor when writing a program. In this paper we present an Information System Prototype developed to store and visualize worked examples derived from transcribed verbal protocols during the process of writing introductory level programs. Empirical data comparing the grades obtained by two groups of novice programming students, using ANOVA, indicates a statistically positive difference in performance in the group using the tool, even though these results still cannot be extrapolated to general population, given the reported limitations of this study.**

***Index Terms*—Basic programming, Verbal Protocols, Dual Coding, e-learning.**

I. INTRODUCTION

A. The programming context

The difficulties many undergraduate students face when learning programming are still a common topic in cognitive, educational and technological research literature. The problem has been approached from many angles, such as the study of the cognitive behavior of novices and experts [1-3] some creative pedagogical strategies [4-6], the cultural environment of the student [7],[8], and of course the use of software tools [9-12].

In developing countries such as Mexico, programming skills are relevant for undergraduate students, given the increasing trends of first world economies to outsource programming, information technology and software related jobs [13-15].

But the most visible aspect of the problem is the almost universal pattern of high failure rates among first year computer science programming students. Depending on the source, it can be found that this failure rates range from 30% to 60% [16-19]

B. The traditional teaching model

Teaching programming is often based on the pedagogical pattern of: 1) presenting the topic, 2) showing a few examples, and 3) assigning practice exercises; that is, the *presentation-examples-practice formula* [20]. And so, a traditional programming course is mainly based on theoretical lecture sessions and practical work on computer laboratories, where most of the content is focused on the characteristics of the computer language being taught [21]. Refs. [22-24] agree that most introductory programming courses are reasonably good to emphasize syntax comprehension of programs, but that they do not reinforce the strategic kind of knowledge required to write programs.

A common perception of computer programming educators is the assumption that this strategic knowledge will develop itself as a byproduct of curricular design [25], while literature suggests that a more effective approach is that this knowledge has to be explicitly taught [21],[26].

C. Software tools

During the last four decades, researchers and designers have been trying to make programming more appealing to students and to the public. They have developed a wide variety of software applications, to make programming skills easier to acquire. From Logo [27], to Alice [11] a diversity of learning goals have been pursued: to develop problem resolution abilities, to develop logical thinking through games, or to facilitate the transition to general purpose programming languages by way of alternative and easier to use interfaces, among other goals. Ref. [28] did a survey of approximately 80 software tools designed to teach programming or to foster the interest in programming by way of games, animations or puzzles.

Other recent types of software tools designed to be an aid in teaching programming are: a) *program visualization*; by using graphics that enable the student to visualize the behavior of algorithms and data structures [29], b) *learning objects*, small instructional components that can be reused in several contexts [10], c) *concept maps*, that work like big knowledge "scaffolds" to represent the main concepts of programming, and to combine them with other teaching strategies and tools [30],[31], and d) *cognitive tutors*, that use declarative and procedural knowledge in

the form of rules, to give guided feedback to the student [32],[33].

All these types of tools have reported positive results, and have had various degrees of success in the goal of teaching programming, but we argue that while some of them have been adopted by programming educators, most of them have focused only on a limited subset of the cognitive aspects (e.g. the transfer of mental models through graphics or interactive feedback) that cognitive literature report as critical.

D. Critical cognitive aspects

The act of programming is essentially a cognitive process of problem resolution that involves writing abstract structures of an algorithmical process. In other words, programming is a way to mentally create a solution to a problem, simultaneously combining a limited and predefined set of syntactic structures and statements, by way of a computer language.

Cognitive literature regarding the acquisition of programming skills is vast and complex. The subject was of special interest in the 1980s [34-36], and in recent years it has still been explored, so that other facets of the problem have been identified and more alternative solutions explored [25],[37],[38]. The most recurring topics found in relevant cognitive literature are: a) comparative studies of mental models of novices and experts, b) the development of programming strategies (also called, plans, schemas or clichés) for common types of problems, and more recently, c) the cognitive process called Metacognition.

1) Mental models

Ref. [3] defines a mental model as an internal representation of a system or complex task, whose construction enables the learner to comprehend and predict the behavior of that system or task.

Ref. [39] also tells that a mental model develops and refines through time, as a result of interaction between the subject and the target system, and that this mental model does not have to be very precise, as long as it is "functional".

In programming, a mental model refers to the image the programmer has about the invisible processing that occurs inside the computer, in the interval between an input and an output [35]. Ref. [40] clarifies that to write a program a person has to have many and very diverse mental models, referring, for example, as to how a loop, a data structure or decision structure behaves. Ref. [41] notes that the existence of a wide range of valid mental models is critical for the novice to acquire the ability to write programs, and if these mental models are not explicitly taught, the student will anyway create its own, of dubious quality and effectiveness.

2) Strategies

It has been found that, even though a student is in fact able to acquire valid mental models, and knows the correct syntax of a programming language, a key cognitive element is still necessary for him to write effective programs. This component is called "strategy" [42] (also known as schema, plan or cliché). Strategies are predefined solutions to stereotyped kinds of problems. The lack of a minimum amount of these strategies restricts the student ability to recognize certain types of problems, and therefore their solution. Ref. [43] indicates that an important aspect of strategies is that *they cannot be deducted from the final form of the program*. This means that a novice can study the final shape of a program, but unless explicitly taught by a teacher, he or she cannot see the process and strategies involved in its writing. The final form of a program can give the student information about the concepts and syntactic structures used, but not about the strategies and decisions applied during the writing process. These strategies are a lot more difficult to teach in the classroom and laboratories, but Ref. [44] notes that "in programming, there is considerable empirical evidence that suggests that strategies are the main basic cognitive component used in design and program comprehension."

Finally, Ref. [45] argues that the process of writing a program does not have to be understood as a "literal transcription" of a previously stored and typified solution, but rather as an iterative, exploratory, and incremental process determined by minor episodes of problem solving and constant re-evaluation of the effectiveness of applied strategies. That is, the effectiveness of a set of strategies is constantly monitored and evaluated by the programmer, in the process of writing the program.

This finding leads to another important aspect of programming (and of problem solving in general) called Metacognition.

3) Metacognition

Ref [46] described Metacognition as "awareness of a person's own cognitive process". While strategies allow a programmer to solve problems, Metacognition allows him to monitor its progress, apply his knowledge to new situations, and identify its own limitations. Ref. [47] indicates that through Metacognition a student can define the nature of a problem or task, select a useful mental representation, use the most pertinent strategy to implement it and put attention to feedback as to how he or she is making progress towards the solution.

In this context, favorable results have been reported through the use of instructional strategies such as "pair programming" [48] (that is, a pair of novice programmers monitor each other's progress, with constant feedback), and with the use of "think-alouds" [49], (instructing the students to verbally reproduce their thought process when writing a program, thus explicitly making such students aware of the decisions, and problem solving strategies they are applying). These are clear examples of Metacognition in the programming context.

Given that empirical evidence in cognitive literature suggests that these three cognitive components (valid Mental Models, Strategies and Metacognition) are critical to acquire the ability to program, we argue that a software tool designed to help students to learn to program, has to include some form of these elements.

II. DEVELOPMENT OF A VERBAL PROTOCOL VISUALIZER TOOL.

A. Verbal protocols

Verbal protocols, as a method of representation and analysis of a person's thought processes, have a solid tradition in the context of cognitive psychology [50-52]

As a technique, verbal protocols were initially developed to study a person's short term memory processes (to what things he or she pays attention to, and in what order, when given a certain task?), but in time, they have been extensively used in other disciplines such as software

engineering (e.g. usability studies [53],[54], software task analysis [55]), and even in programming teaching [49].

To develop our tool, we selected the method of verbal protocols, as a way to elicit (and later explicitly show to students) an expert programmer's series of decisions when given a certain basic programming type of problem: that is, a verbal protocol can show a student what elements of a problem the expert is paying attention to, how and in what form the programmer applies the basic programming structures (loops, decisions, data structures), and how does the programmer identifies when he or she made a mistake and has to backtrack and correct it.

To analyze a verbal protocol, a researcher has to rely on some kind of recording device (in the old days, a tape recorder) to be able to transcribe, apply a coding scheme and compare the verbalizations of a given set of subjects. At this point we opted for a different kind of recording method, using *video capture software*, to be able not only to record the verbal data, but also *the visual behavior* of the expert programmer.

For example, in our study, a recording session would consist of asking an expert programmer to write a program to solve a simple programming problem, such as the following, while verbalizing his or her thought process:

Write a program in C Language that, when given a quantity N of integers, gives the sum of all pair numbers, and the average of all uneven numbers.

Then, we used the video capture software to record audio and video activity taking place in the computer. The resulting product was a video file with visual and audio information that was later transcribed and edited to a database (Figure 1)

It has to be noted that, in all cases, video editing was needed to re-record the video segments of the protocols, because correspondence between the verbalizations of the programmer, and the visual information (the actual writing of the code) where very rarely synchronized (Figure 2)

For our experimental test, four edited protocols where produced, representing four types of problems with different levels of difficulty (Figure 3).

Once the tool was in its final form, students could access and visualize the protocols using a web based interface, typing a keyword, an specific phrase or an author's (programmer) name (Figure 4).

B. *Dual coding*

Dual coding theory (DCT) describes that to process sensorial stimuli from the environment, the human mind has two independent but connected memory subsystems: one for visual and one for verbal information. The visual subsystem handles concrete images and sounds. The verbal subsystem records language and abstract information. According to the theory, both systems function independently but are intimately connected: when a verbal representation is created as a response to a visual image, or when an image is created as a result of seeing or hearing a word, it is said that a *referential connection* has been made, and thus, *dually coded*. [56-58].

Empirical data of DCT studies [56],[57],[59] shows that the brain can retrieve information better when it is dually coded.

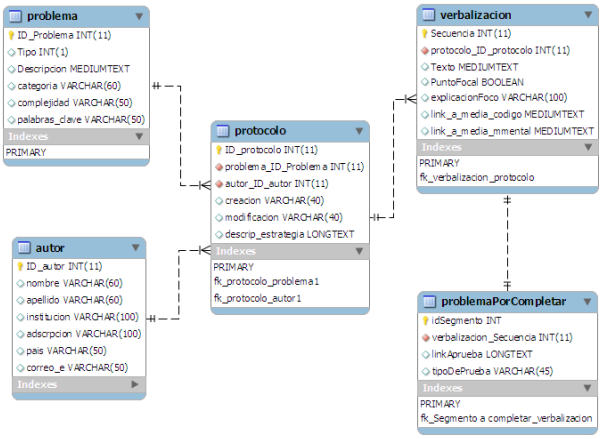

Figure 1. Data Model used to store and retrieve verbal protocols

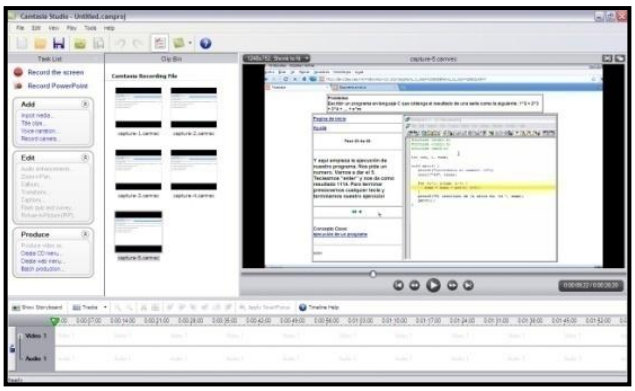

Figure 2. Video editing of verbalization segments of protocols

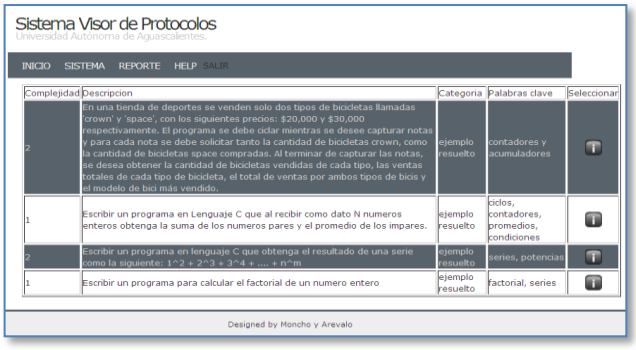

Figure 3. List of protocols available in the tool

Figure 4. Searching interface

In our study, we tried to apply this principle to the design of the user interface of the tool, by allowing the student to browse the verbal and visual information of the protocols (Figure 5). The protocols were divided in segments that students could study and analyze by reading the verbalizations, and watching the video segment corresponding to the writing of the code.

The interface was subject to several tests to further refine its usability. For example a feature was added to allow a student to "jump" directly to a specific step of the protocol (Figure 6).

C. Experimental conditions

To measure the tool's capability to transfer strategic knowledge to novice undergraduate programmers, we designed a standard test, consisting of three basic programming problems, to assess the student's skills. The test was written to evaluate the following specific abilities:

a) Recognize types of problems that involved combined structures of repetition (loops) and selection (if).
b) Effectively write repetition and decision structures.
c) Recognize and effectively apply problems that involve counters.
d) Make calculations involving exponents.

It has to be noticed that the test was designed using previously applied questions and problems, taken from our internal programming academy quiz repository. This repository of tests dates back to the year 2008. The specific sample of the three test questions was randomly selected to design the measurement instrument, but taking into account their similarity to the desired specific abilities to be measured. 15 historical undergraduate student's results of both computer science and electronic engineering were selected.

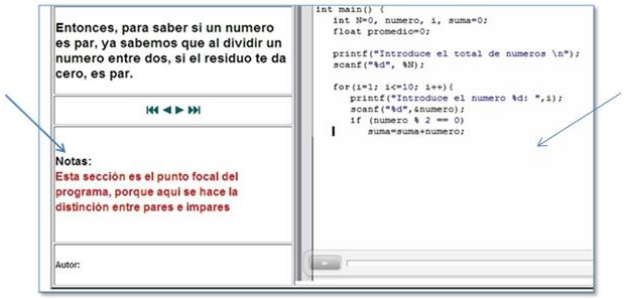

Figure 5. Dual coding user interface.

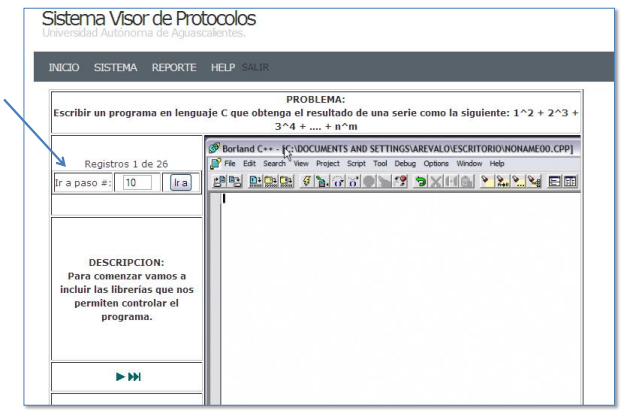

Figure 6. Usability modifications to user interface

These previous results were analyzed to verify if the instrument's behavior was normal and without significant bias. We used a grading scale of 0 to 10.

Descriptive statistics (Table I), normality tests (Table II**Fehler! Verweisquelle konnte nicht gefunden werden.**), the corresponding histogram of the instrument data (Figure 7), and a Q-Q Plot (Figure 8) are shown.

Given the small size of the sample we look at the Shapiro-Wilk test for normality asumption. In this case, the Sig. value is greater than 0.05, wich indicates that the data is normal.

Also, in Figure 8, we can see that the data obtained with the instrument (that is, the grades obtained) also behave normally, except for one observed value.

TABLE I.
MEASUREMENT INSTRUMENT DESCRIPTIVE STATISTICS

		Grade
N	Valid	15
	Mean	5.7340
	Median	5.8300
	Mode	8.83
	Std. Deviation	2.27753
	Variance	5.187
	Skewness	-0.679
	Std. error of Skewness	0.580
	Kurtosis	0.879
	Std. error of kurtosis	1.121
	Range	8.50
	Minimum	0.33
	Maximum	8.83

TABLE II.
NORMALITY TESTS

Instrument	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Statistic	*df*	*Sig.*	*Statistic*	*df*	*Sig.*
Grade	.129	15	.200*	.939	15	.370

*. This is a lowerbound of true significance.
[a] Lilliefors Significance Correction.

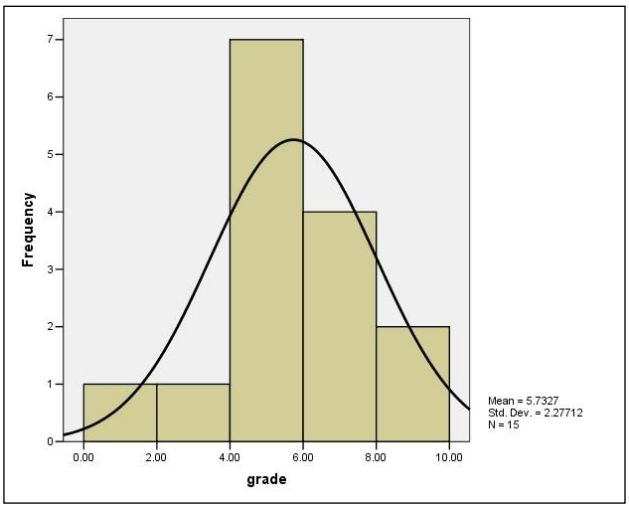

Figure 7. Measurement instrument histogram

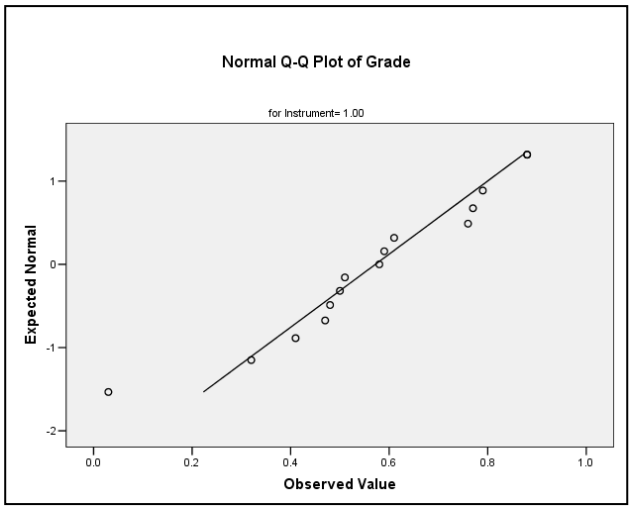

Figure 8. Q-Q Plot of grades obtained with instrument

To test the effectiveness of our tool, a semi-experimental setting was designed, using two groups of programming students: one from 2nd semester Computer Science students and another from 2nd semester Electronic Engineering students, both from Autonomous University of Aguascalientes (UAA), México. Selection of participants was not random. Complete groups were invited, given that the experiment was conducted during a period of normal classes.

The Electronic Engineering group (n=20) was selected as the control group (TRAD). The computer science group (n=18) was to serve as the experimental group (EXP). 55% of the (TRAD) group had previous programming experience from highschool, while 41% of the (EXP) group had previous programming experience.

Two days before the application of the test, four excersices were given to both groups for them to study and practice. This excersices were the same ones loaded in the Protocol Visualizer Tool.

Our research model had as independent variable the "teaching method", so that the control group (TRAD) had a teacher giving a traditional lecture using blackboard and laboratory computers to explain the solving procedure of the given practice excersices, and the experimental group (EXP) used the Protocol Visualizer Tool to study the solving procedure of those same excersices. Our dependent variable was "performance": that is, the grade obtained through the instrument.

Controlled conditions for both groups were:

- Explanatory lecture sessions had a one hour duration for both groups.
- The given time to answer the instrument was limited to an hour for both groups.
- A "motivation" factor was introduced for both groups in form of "extra points".
- Characteristics of the lab computers used from both groups were the same.
- Previous programming experience was similar in both groups (55% for TRAD and 41% for EXP).
- At the time of the experiment, the instructional content given (during normal classes) to both groups was the same. Both groups where studying basic data structures in C Language.

Uncontrolled conditions were:

- Teachers from both groups were different, but they came from the same programming academy, taught the same content, and had the same experience in teaching.
- It was not possible to record the individual answering time of the test participants.
- Selection of participants was not random. In both cases, complete groups where invited to participate in the study.

III. RESULTS

A. *Descriptive statistics*

The test was graded by professors of our internal programming academy staff.

Table III shows the results of the descriptive statistics of both groups. Mean results for EXP group was 6.33 and 3.79 for TRAD group. The Median value for EXP group was 7.5, and 3.7 for the TRAD group. Mode for the EXP group was 3.30 and 2.30 for the TRAD group. Mean values indicate that the experimental had performed 25% better than the control group; but Modes results suggests that both groups had bad performers. Standard deviation of the EXP group was 2.99, which indicates bigger dispersion of data than the TRAD group (2.35). This result indicates that the TRAD group performed "uniformly bad", and that the EXP group had more "better than average" performers.

Comparative histograms of both groups' results are shown in Figures 8 and 9 . It can be seen that EXP group is negatively skewed to the left, meaning that the majority of its frequencies are grouped towards the upper values of the scale. TRAD group frequencies shown in Figure 9 show the opposite behavior.

B. *ANOVA*

Given the descriptive statistics results, it can be inferred that the EXP group had better performance than the TRAD group. To see if this performance was statistically significant, we ran an ANOVA test with the results shown in Table IV.

These results indicate that there was a statistically significant different performance (p<=.006) between the two groups.

TABLE III.
COMPARATIVE DESCRIPTIVE STATISTICS OF GROUP PERFORMANCE

		Grade (EXP group)	Grade (TRAD group)
N	Valid	18	20
	Mean	6.3389	3.7900
	Median	7.5000	3.7000
	Mode	3.30(a)	2.30
	Std. Deviation	2.99257	2.35683
	Variance	8.955	5.555
	Skewness	-.499	0.689
	Std. error of skewness	0.536	0.512
	Kurtosis	-1.412	0.443
	Std. error of kurtosis	1.038	0.992

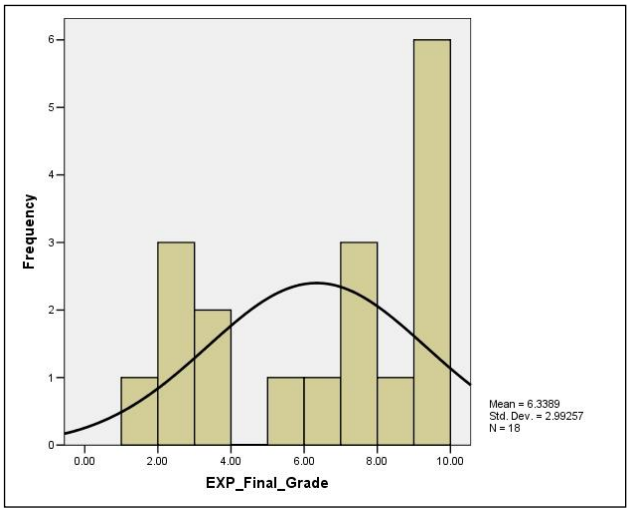

Figure 9. Test results of EXP group

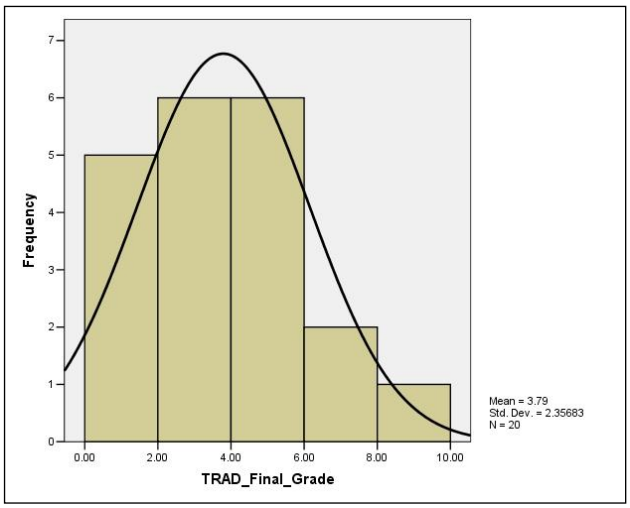

Figure 10. Test results of TRAD group.

TABLE IV.
ANOVA TEST OF "PERFORMANCE" VARIABLE FOR TRAD AND EXP GROUP

	Sum of squares	df	Mean square	F	Sig
Between groups	61.549	1|	61.549	8.596	0.006
Within groups	257.781	36	7.161		
Total	319.330	37			

C. Correlations

Programming literature suggests that there is a positive correlation between previous experience and performance in the first undergraduate programming course [37],[60],[61]. We ran a correlation test to see if a correlation could exist between previous experience of these participants and the results obtained in the test. As mentioned earlier 55% of TRAD group participants have had previous experience in programming, and 41% of EXP participants had studied programming in high school. Pearson correlation results are shown in V and Table VI.

TABLE V.
EXP GROUP PEARSON CORRELATION.

		Previous experience	Grade
Previous experience	Pearson correlation	1.000	0.334
	Sig. (2-tailed)		0.175
	N	18	18
Grade	Pearson correlation	0.334	1.000
	Sig. (2-tailed)	0.175	
	N	18	18

TABLE VI.
TRAD GROUP PEARSON CORRELATION.

		Previous experience	Grade
Previous experience	Pearson correlation	1.000	-0.139
	Sig. (2-tailed)		0.558
	N	20	20
Grade	Pearson correlation	-0.139	1.000
	Sig. (2-tailed)	0.558	
	N	20	20

Pearson results show that, for both groups, there is no significant correlation between their previous programming experience, and their performance in the study.

IV. CONCLUTIONS

A. Explicit strategy learning

These positive results are promising in the sense that, under the conditions of the study, a significant improvement (25%) in performance was obtained.

It can be interpreted that when students where using the tool and explicitly studied (by reading and seeing) the problem solving procedure, they learned a small set of strategies that where effective to those kinds of problems.

Also, the verbalization feature of the protocols, allowed the students to understand why the expert programmer was using a specific kind of programming structure.

In the protocols, a metacognitive element was implicitly present during some "backtracking" episodes. For example: in one case, a programmer identified that she had omitted the declaration of a variable, and that was causing a syntax error in the program. In other case, (in the final steps of the protocol) the programmer noted that she needed a counter variable to obtain the average of the uneven numbers. These incidents showed the students that the process of writing a program is not linear, but incremental and constantly monitored.

Our results also seem to support previous studies related to some instructional strategies in the context of cognitive load theory [62]. That is, that showing a "worked example" [63-65] can be an effective instructional strategy, given the reduced "memory load" that the student is submitted to.

B. Benefits

We argue that the tool can be helpful in decreasing high failure rates, if the amount of protocols loaded in the tool is sufficiently big to cover a significant range of problem categories. That is, if students are given a wider range of problem solving strategies, the transit between the initial states of learning [66] and towards an automation of strategies can be made more efficiently.

Also, programming teachers can share their knowledge with a wider range of students, whom, in turn, can constructively compare different kinds of solutions for one same type of problem.

It is important to note that the tool was designed as an aid and complement to programming teachers, and not as a substitute to them.

C. *Limitations.*

Given the uncontrolled conditions reported in previous sections of this paper, the positive results obtained by the EXP group using the Protocol Visualizer Tool, cannot be generalized to be valid for all the population of first year undergraduate programming students.

An uncontrolled variable was teaching style: that is, even though both teachers were part of the same academy and had similar background and experience, the two groups having different teachers could have had an unmeasured effect on the final results.

Also, duration of the actual study was limited to three days, given the limited availability of the students who voluntarily participated.

D. *Future studies*

Future studies need to be longitudinal in nature, so that the effect of a longer exposure of the students to the tool can be measured.

A randomly selected sample of participants is desirable, but this kind of scenario is not always possible (or practical) given the nature of every day lectures.

It is also possible to load the tool with protocols that involve problems related to other programming languages or paradigms such as Java, C#.

Also, the graphical user interface can still be improved through usability test, in order to use within other Web-enabled platforms (such as mobile devices).

Lastly, it is planned to extend the functionality of the tool, by adding pedagogical features such as *completion problems*, in selected segments of the protocols, to be in accordance to suggestions given by Refs [67],[68].

REFERENCES

[1] V. Fixx, S. Wiedenbeck, y J. Scholtz, "Mental Representations of Programs by Novices and Experts," Amsterdam, The Netherlands: ACM Press New York, NY, USA, 1993, pp. 74-79.

[2] L. Ma, J. Ferguson, M. Roper, y M. Wood, "Investigating the viability of mental models held by novice programmers," *ACM SIGCSE Bulletin*, vol. 39, 2007, pp. 499-503. doi:10.1145/1227504.1227481

[3] C.E. George, "Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models," A.F. Blackwell & E. Bilotta, 2000.

[4] V. Dagdilelis, "Introducing Secondary Education Students to Algorithms and Programming," *Education and Information Technologies*, vol. 9, 2004, pp. 159–173. doi:10.1023/B:EAIT.0000027928.94039.7b

[5] D. Hagan, "The value of discussion classes for teaching introductory programming," Dublin City Univ., Ireland: ACM Press New York, NY, USA, 1998, pp. 108 - 111.

[6] T. Jenkins, "A participative approach to teaching programming," Dublin City Univ., Ireland: ACM Press New York, NY, USA, 1998, pp. 125 - 129.

[7] C. Bruce, "Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university," *Journal of Information Technology Education*, vol. 3, 2004, pág. 144.

[8] S. Booth, "Learning to program as entering the datalogical culture: a phenomenographic exploration," Fribourg Switzerland, August 2001: 2001.

[9] L. McIver, "GRAIL: A Zeroth Programming Language," Chiba, Japan: 1999, pp. 43-50.

[10] E. Kujansuu, "Using program visualisation learning objects with non-major students with different study background," Tampere, Finland: 2006, pp. 21-26.

[11] M.J. Conway, "Alice:Easy-to-Learn 3D Scripting for Novices," PhD. Thesis, University of Virginia, 1997.

[12] J. McKeown, "The use of a multimedia lesson to increase novice programmers' understanding of programming array concepts," *Journal of Computing Sciences in Colleges*, vol. Volume 19, Abr. 2004, pp. 39 - 50.

[13] M. de Raadt, R. Watson, y M. Toleman, "Language tug-of-war: industry demand and academic choice," Adelaide, Australia: 2003, pp. 137 - 142.

[14] K.S. Koong, L.C. Liu, y X. Liu, "A Study of the Demand for Information TechnologyProfessionals in Selected Internet Job Portals," *Journal of Information Systems Education*, vol. 13, 2002, pp. 21-28.

[15] C. Litecky, B. Prabhakatar, y K. Arnett, "The IT/IS job market: a longitudinal perspective," Claremont, California, USA: 2006, pp. 50-52.

[16] T. Boyle, "Improved success rates for students studying Programming," *Investigations in university teaching and learning*, vol. 1, 2003, pp. 52-54.

[17] M. Guzdial, "Teaching the Nintendo generation how to program," *Communications of the ACM*, vol. 45, Abr. 2002, pp. 17-21. doi:10.1145/505248.505261

[18] V. Aleven y K. Koedinger, "An effective metacognitive strategy: learning by doing and explaining with a computer-based Cognitive Tutor," *Cognitive Science*, vol. 26, 2002, pp. 147–179. doi:10.1207/s15516709cog2602_1

[19] D.D.E.I. UAA, *Porcentajes de reprobación en la materia de Programación I, en las carreras de LI-LTI/ISC durante los años 2004 y 2007*, Aguascalientes: Universidad Autónoma de Aguascalientes, 2007.

[20] S. Shaffer, "Ludwig: An online programming tutoring and assessment system," *Inroads - The SIGCSE Bulletin*, vol. 37, 2005, pp. 56-60. doi:10.1145/1083431.1083464

[21] A. Robins, J. Rountree, y N. Rountree, "Learning and Teaching Programming: A Review," *Computer Science Education*, vol. 13, 2003, pp. 137–172. doi:10.1076/csed.13.2.137.14200

[22] M. Linn y M. Clancy, "The Case for Case Studies of Programming Problems," *Communications of the ACM*, vol. 35, 1992.

[23] M. Linn, "The Cognitive Consequences of Programming Instruction in Classrooms," *Educational Researcher*, vol. 14, 1985.

[24] T. McGill y S. Volet, "A Conceptual Framework for Analyzing Students' Knowledge of Programming," *Journal of Research on Computing in Education*, vol. 29, 1997.

[25] J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. St.Clair, y L. Thomas, "A cognitive approach to identifying measurable milestones for programming skill acquisition," Bologna, Italy: ACM New York, NY, USA, 2006, pp. 182 - 194.

[26] S. Volet y C. Lund, "Metacognitive instruction in introductory computer programming: A better explanatory construct for performance than traditional factors," *Journal of educational computing research*, vol. 10, 1994.

[27] S. Papert, *Mindstorms, children, computers and powerful ideas*, Basic Books, New York., 1980.

[28] C. Kelleher y R. Pausch, "Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers," *ACM Computing surveys (CSUR)*, vol. 37, 2005, pp. 83 - 137.

[29] S. Pollack, "Selecting a Visualization System," Warwick, UK: 2004.

[30] A. Maries y A. Kumar, "Concept maps in intelligent tutors for programming," *Journal of Computing Sciences in Colleges*, vol. 22, 2007, pág. 54.

[31] A.N. Kumar, "Using Enhanced Concept Map for Student Modeling in Programming Tutors," Melbourne Beach, Florida: 2006, pp. 527-532.

[32] J.R. Anderson, A. Corbet, y K.R. Koedinger, "Cognitive Tutors, Lessons Learned," *The Journal of Learning Sciences*, vol. 4, 1995, pp. 167-207. doi:10.1207/s15327809jls0402_2

[33] V. Aleven, B. McLaren, J. Sewall, y K. Koedinger, "The Cognitive Tutor Authoring Tools (CTAT): Preliminary evaluation of efficiency gains," Maceio, Brazil: 2006.

[34] J.R. Anderson, F. Conrad, y A. Corbett, "Skill Aquisition and the Lisp Tutor," *Cognitive Science*, vol. 13, 1989, pp. 467-505. doi:10.1207/s15516709cog1304_1

[35] P. Bayman, "A diagnosis of beginning programmers' misconceptions of BASIC programming statements," *Communications of the ACM*, vol. 26, Sep. 1983, pp. 677 - 679. doi:10.1145/358172.358408

[36] J. Larkin y H. Simon, "Why a diagram is (sometimes) worth ten thousand words," *Cognitive Science*, vol. 11, 1987, pp. 65-99. doi:10.1111/j.1551-6708.1987.tb00863.x

[37] V. Ramalingam, "Self-Efficacy and mental models in learning to program," Leeds, United Kingdom: ACM Press New York, NY, USA, 2004, pp. 171 - 175.

[38] S. Wiedenbeck, D. LaBelle, y V. Kain, "Factors affecting course outcomes in introductory programming," Institute of Technology, Carlow, Ireland: 2004.

[39] D. Norman, *Some observations on mental models.*, Erlbaum, Hillsdale, NJ., 1983.

[40] P. Johnson-Laird, *Mental Models*, Cambridge University Press, 1983.

[41] L. Winslow, "Programming Pedagogy - A psychological Overview," *SIGCSE Bulletin*, vol. 28, 1996, pp. 17-22. doi:10.1145/234867.234872

[42] R. Rist, "Learning to program: schema creation, application and evaluation," *Computer Science Education and Research*, Netherlands: Taylor & Francis Group, 2004, pp. 175-197.

[43] R. Brooks, "Categories of programming knowledge and their application," *International Journal of Man-Machine Studies*, vol. 18, 1990, pp. 543-554. doi:10.1016/S0020-7373(83)80031-5

[44] R. Rist, "Program Structure and Design," *Cognitive Science*, vol. 19, 1995, pp. 507-562. doi:10.1207/s15516709cog1904_3

[45] S. Davies, "Models and Theories of programming strategy," *International Journal of Man-Machine Studies*, vol. 39, 1993, pp. 237-267. doi:10.1006/imms.1993.1061

[46] J. Flavell, "Metacognition and cognitive monitoring: A new area of cognitive developmental inquiry," *American Psychologist*, vol. 34, 1979, pp. 906–911. doi:10.1037/0003-066X.34.10.906

[47] A.F. Gourgey, "Metacognition in basic skills instruction," *Instructional Science*, vol. 26, 1998, pp. 81-96. doi:10.1023/A:1003092414893

[48] L. Williams y R.L. Upchurch, "In support of student pair-programming," *ACM SIGCSE Bulletin*, vol. 33, 2001, pp. 327 - 331. doi:10.1145/366413.364614

[49] N. Arshad, "Teaching Programming and Problem Solving to CS2 Students using Think-Alouds," *ACM SIGCSE Bulletin*, vol. 41, 2009, pp. 372-376. doi:10.1145/1539024.1508998

[50] K. Ericsson y H. Simon, *Protocol Analysis. Verbal reports as data*, Cambridge Massachusets.: MIT Press, 1993.

[51] A. Newell y H. Simon, *Human Problem Solving*, Englewood Cliffs, NJ: Prentice Hall, 1972.

[52] J. Russo, E. Jonson, y D. Stephens, "The validity of verbal methods," *Memory Cognition*, vol. 17, 1989, pp. 759-769. doi:10.3758/BF03202637

[53] S. Knox, W. Bailey, y E. Lynch, "Directed dialogue protocols: verbal data for user interface design," ACM New York, NY, USA, 1989, pp. 283 - 287.

[54] J. Nielsen, T. Clemmensen, y Y. Carsten, "Getting access to what goes on in people's heads?: reflections on the think-aloud technique," Aarhus, Denmark: ACM New York, NY, USA, 2002, pp. 101 - 110.

[55] I. Vessey y S. Conger, "Requirements Specification: Learning Object, Process, and Data Methodologies," *Communications of the ACM*, vol. 37, 1994.

[56] M.A. Kuo y S. Hooper, "The Effects of Visual and Verbal Coding Mnemonics on Learning Chinese Characters in Computer-Based Instruction," *Educational Technology Research and Development*, vol. 52, 2004, pp. 23-34. doi:10.1007/BF02504673

[57] R.E. Meyer y V.K. Sims, "For Whom Is a Picture Worth a Thousand Words? Extensions of a Dual-Coding Theory of Multimedia Learning," *Journal of Educational Psychology*, vol. 86, 1994, pp. 389-401. doi:10.1037/0022-0663.86.3.389

[58] M. Sadoski y A. Paivio, "A Dual Coding Theoretical Model of Reading," *Theoretical models and processes of reading*, Newark, DE: International Reading Association, 2004.

[59] J. Kounios y P. Holcomb, "Concreteness effects in semantic processing: ERP evidence supporting dual-coding theory.," *Journal of Experimental Psychology: Learning, Memory and Cognition*, vol. 20, 1994, pp. 804-823. doi:10.1037/0278-7393.20.4.804

[60] D. Hagan y S. Markham, "¿Does it help to have some programming experience before beginning of a computer degree program?," Helsinki, Finland: 2000, pp. 25 - 28.

[61] A.F. Blackwell, "First Steps in Programming: A Rationale for AttentionInvestment Models," Arlington, VA: 2002, pp. 2-10.

[62] J. Sweller, J. van Merrienboer, y F. Paas, "Cognitive Architecture and Instructional Design," *Educational Psychology Review*, vol. 10, 1998, pp. 251-295. doi:10.1023/A:1022193728205

[63] M. Ward y J. Sweller, "Structuring effective worked examples," *Cognition and Instruction*, vol. 7, 1990, pp. 1-39. doi:10.1207/s1532690xci0701_1

[64] J. Sweller y G. Cooper, "The use of worked examples as a substitute for problemsolving in learning algebra," *Cognition and Instruction*, vol. 2, 1985, pp. 59-89. doi:10.1207/s1532690xci0201_3

[65] M.E. Caspersen y J. Bennedsen, "Instructional design of a programming course: a learning theoretic approach," Atlanta, Georgia, USA: ACM New York, NY, USA, 2007, pp. 111 - 122.

[66] J.R. Anderson, J. Fincham, y S. Douglass, "The role of examples and rules in the acquisition of a cognitive skill," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 23, 1997, pp. 932-945. doi:10.1037/0278-7393.23.4.932

[67] J. van Merrienboer y H. Krammer, "The "completion strategy" in programming instruction: Theoretical and empirical support," *Research on Instruction*, Englewood Cliffs, NJ: Educational Technology Publications, 1990, pp. 45-61.

[68] J. van Merrienboer, "Strategies for programming instruction in high school: Program completion vs. program generation," *Educ. Comput. Res.*, vol. 6, 1990, pp. 265-287. doi:10.2190/4NK5-17L7-TWQV-1EHL

AUTHORS

Carlos A. Arévalo, Ph.D., is a full time research professor at the Department of Information Systems within the Autonomous University of Aguascalientes, México. He earned his Doctoral degree in Software Engineering in the same University, and specializes in e-learning technologies (e-mail: carevalo@correo.uaa.mx).

Estela L. Muñoz, Ph.D., is a full time research professor at the Department of Electronic Systems within the Autonomous University of Aguascalientes, México. She is an experienced teacher of programming and data structures. (e-mail: elmunoz@correo.uaa.mx).

Juan M. Gómez, Ph.D., is a full time research professor, at the Department of Electronic Systems within the Autonomous University of Aguascalientes, México. He has made contributions in the field of design research (e-mail: jmgr@correo.uaa.mx).

The work presented in this paper will be extended and continued with support and as part of a three year European Commission ALFA III project grant (**DCI-ALA/19.09.01/21526/245-315/ALFAHI(2010)/123), IGUAL "Innovation for Equality in Latin American University"** beginning December 2010.

This article is an extended version of a paper presented at the Interdisciplinary Conference of AHLiST 2010 Conference, June 2010, Madrid, Spain. Received May 10th, 2011. Published as resubmitted by the authors on June 9th, 2011.