Coached Program Planning: Dialogue-Based Support for Novice Program Design

H. Chad Lane and Kurt VanLehn
Department of Computer Science
Learning Research and Development Center
University of Pittsburgh
{hcl,vanlehn}@cs.pitt.edu

Abstract

Coached program planning is a dialogue-based style of tutoring aimed at helping novices during the early stages of program writing. The intent is to help novices understand and solve problems in their own words through the construction of natural-language style pseudocode as the first step in solving a programming problem. We have designed an environment supporting coached program planning and have used it in a human-to-human, computer-mediated evaluation of 16 novice programmers enrolled in a pre-CS1 programming course at the University of Pittsburgh. The results show that students who underwent coached program planning, compared to those who did not, were more prolific with comments in their programs, committed fewer structural mistakes, and exhibited less erratic programming behavior during their implementation. The dialogues collected from this experiment followed a clear 4-step pattern. Starting with this observation, we are developing a dialogue-based intelligent tutoring system called the Pseudocode Tutor to support coached program planning.

Categories & Subject Descriptors

K.3 *Computers & Education*: Computer and Information Science Education - Computer Science Education

General Terms

Algorithms, Design, Human Factors

Keywords

novice programming, structured programming, intelligent tutoring systems, coached program planning, dialogue systems

SIGCSE '03, February 19-23, 2003, Reno, Nevada, USA.

1 Introduction

Nearly every aspect of writing a program can be a struggle for a novice, and the situation is worsened by attempts to deal with multiple impasses at once [4]. Additionally, novices spend little time planning their code before keying it in [10] and have a tendency to believe a program that produces correct answers is all that matters [7]. Of course, novices rarely produce perfect programs in their early attempts, especially if no planning was done. Soloway and Spohrer have found that many of the bugs they encounter reflect misconceptions about the underlying algorithm rather than the language being used [14].

In an effort to make programming less difficult for beginners, a number of researchers have advocated the design of programming languages that better match natural ways of thinking and communicating [9, 2]. The idea is to "close the gap" between a problem statement and a programmed solution by limiting the cognitive hurdles imposed by traditional programming languages, thereby making it easier for novices to express their solutions in the desired language. A slightly different approach, but also based on finding common ground in language with novices, was adopted in BRIDGE [1]. This system helped students to construct a solution description in natural language, followed by successive rewrites in more precise forms, ultimately resulting in a working program. The idea of having students explain and elaborate ideas in their own words is a known effective method for improving learning [3]. It is unclear if these benefits were reaped in BRIDGE because the natural language solutions were created via menu selections.

In this paper, we introduce *coached program planning* (CPP), a dialogue-based style of tutoring aimed at helping novices during the earliest stages of programming. We show an environment supporting CPP and report the results of a human-to-human evaluation revealing that students who used it exhibit more desirable behaviors during programming than students who did not. We conclude with a discussion of the results, and a look forward to the ultimate goal of this research: creation of an intelligent tutoring system for CPP.

2 Coached Program Planning

In a CPP tutoring session, the student and tutor collaborate to build a natural-language-style pseudocode solution to a problem. Ideally, the student has already read the problem statement, but has not yet attempted an implementation. Dialogue is the vehicle for this collaboration, and the result-

ing artifact (i.e., the pseudocode) can then be used by the student as a blueprint during the non-tutored implementation phase (using an editor, compiler, etc.).

2.1 Dialogue Structure

A CPP dialogue consists of the tutor repeatedly asking the student to (1) identify a programming goal, (2) describe a technique for attaining this goal, (3) suggest pseudocode steps that attain the goal, and finally (4) place the steps appropriately within the pseudocode. This continues until all programming goals have been satisfied, and the pseudocode is complete. This 4-step pattern governed the dialogues in our corpus (described later). At the tutor's discretion, the student's own words are used as much as possible in the text of the pseudocode steps. It is likely that many of the answers the four questions above will be flawed (partially correct, imprecise, or simply wrong). The collaborative nature of dialogue is ideal for refining these kinds of answers into better ones.

In addition, it is common for the tutor to enter sub-dialogues involving the layout and organization of the pseudocode. These arise from either algorithmic errors by the student (e.g., improperly sequencing steps) or structural issues (e.g., improper indentation).

2.2 Pedagogy

Pseudocode is a well established approach to teaching novice program design. At one extreme, Shackelford promotes the exclusive use of (formal) pseudocode [13], while others have argued for a gradual introduction along with a particular programming language [8]. Our approach is closer to the latter, but with informal pseudocode in an effort to better match the language of the student. The aim of CPP is to instill a mindset in novices of thinking about a problem first in their own words, rather than implementation and programming language specific issues.

Because the tutor acts as a filter for the student's ideas, the resulting pseudocode will be correct. Thus, heading into implementation, students are armed with a better understanding of the problem and a better idea of the overall algorithm. The individual pseudocode steps are implicitly approved and often edited by the tutor before making their way to the solution. Thus, the style of the pseudocode is up to the tutor. The basic requirement is that the student should require minimal (if any) training to generate the text for the steps. In summary, each step is suggested by the student, created by the tutor, and finally placed by the student. CPP is a preventative approach in that program designs with serious mistakes and omissions should be avoided entirely because of the tutor's presence. Of course, mistakes can still be made during implementation, but they should rarely be algorithm related (assuming the student understands the pseudocode). Making serious mistakes and learning from them is essential to maturing as a programmer, however, so we do not advocate using CPP for all assignments in an introductory programming course.[1]

2.3 Examples

In this section, we present a pseudocode sample along with two dialogues (all taken from actual tutoring sessions). The problem in question – the "Hailstone" series – requires the student to produce a sequence of integers according to simple rules. The student is asked to have the program report the number of items in the resulting sequence, as well as the largest value encountered. Figure 1 shows a student's solution to the Hailstone problem. This pseudocode is in its final form, the same as what the student physically takes away from the tutoring session.

```
1   get a positive number from the user
2   initialize count to one
3   initialize largest to the input
4   while number generated is not 4 do
5     if the number is odd then
6           multiply it by 3 and add one
7     else
8           divide the num by 2
9     increase count by 1
10    if current value is larger than largest then
11          store the new largest value
12  end while
13  print count
14  print largest
```

Figure 1: pseudocode from the Hailstone problem

```
0 T what do we need to work on next?
1 S Finding the how many numbers are in the
    sequence.
2 T are you sure? [pause]
3 T does our program generate an entire sequence?
4 S We need to see if the new number is now
    odd or positive.
5 T right... and after we do that, are we done?
6 S Not unless we reach the ground state.
7 T how can that be determined in the program?
8 S by a loop
9 T right!
```

Figure 2: determining the need for a loop

There are several subtleties resulting from the open-endedness of the pseudocode that should be noted. For example, *the input* in line 3 refers to the user input accomplished in line 1. Similarly, the variable holding the changing Hailstone value is referred to in several different ways: *number generated* (line 4), *the number* (5), *it* (6), *the num* (8), and *current value* (10). Also, the steps involving calculations (6 and 8) only implicitly suggest an update. In other words, assignment of a new value is not represented by the step itself. While these rarely pose a problem for a human tutor, they represent significant challenges for an ITS attempting to do natural language understanding.

In the dialogue shown in figure 2, the student seems to lack a heuristic of the form *if a sequence of values needs to be generated, use a loop*, or at least needs help in applying it to this problem. It is interesting to note that the student suggested the loop body (4) and the condition on the loop (6)

[1]Perhaps just the first half or two-thirds.

0 T What condition will we need on that loop?
1 S *while hailnumber is even do*
2 T think about that. you want to repeat the even/odd check over and over.
3 T in the example you did, when did you stop?
4 S *when it went to the same numbers*
5 T yep. that was called the "ground state".
6 T so we'll loop while what is true?
7 S *hailnumber is not equal to 4*
8 T good.

Figure 3: finding the condition on a loop

before mentioning a looping construct at all. This observation is in harmony with previous findings that novices prefer a depth-first search approach to programming [11] and not the breadth-first ("top-down") approach typically taught in introductory programming courses. This student is given encouragement and positive feedback for relevant and important suggestions, but is gently brought to what expert programmers do first: identify higher level goals. Dialogue seems to be an effective medium to achieve this pedagogical goal. It should also be noted that this dialogue does not specifically follow the 4-step pattern: the tutor seems to switch from looking for a programming goal to looking for a technique.

The second dialogue, shown in Figure 3 (and from a different subject), shows the tutor helping the student to determine the termination condition on the loop. Two tutoring strategies are employed here. First, the tutor refers to an example in line 3. Second, in line 6, the tutor poses a completion question. In our experiments, it was common to see the tutor ask the student specify a loop condition only (answered in line 7), and subsequently create the step augmenting with *while* and *do*.

3 An Environment for CPP

We have built an interface supporting CPP for the purpose of evaluation and to act as the front end of an intelligent tutoring system. The interface consists of three windows (figure 4). The mini-browser (upper left) displays HTML pages and remains available throughout the tutoring session. The dialogue window (lower left) allows communication between a human tutor and the student. Finally, the pseudocode window (right half) contains draggable *tiles* that contain text, each of which represents a step in the solution.

A tutoring session begins with the student reading the problem statement in the browser, followed by collaborative pseudocode construction. When steps are agreed on, the tutor creates tiles and the student drags them into the rest of the pseudocode. Upon completion, the tile outlines are removed leaving the pseudocode in a more traditional form for the student to print out and use during implementation.

4 Experiment

In the fall of 2001 and spring of 2002, we conducted a computer-mediated, human-to-human study to assess both the usability of the environment and the effect of CPP on novices. CPP is intended to help novices develop their algorithmic skills, express programming concepts in their own words, and to build pseudocode. Thus, we predicted that students undergoing CPP would (1) show less erratic behavior in dealing with algorithmic difficulties during implementation, (2) be more prolific with their program comments, and (3) make fewer indentation mistakes in their source code.

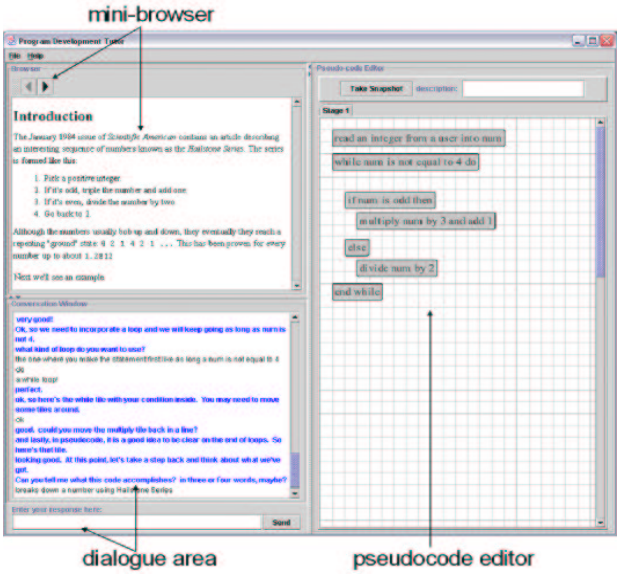

Figure 4: CPP Environment

4.1 Subjects

Subjects were volunteers from *Introduction to Computer Programming*, a pre-CS1 service course at the University of Pittsburgh and were paid $7/hour for their participation. 16 students volunteered and 2 were removed for not completing the course. None of the subjects had programming experience beyond a few weeks of BASIC or spreadsheet skills.

4.2 Procedure

Four projects from the course were used in this study, starting roughly one month into the course. The first assignment required the students to convert a four-digit number into words. The second assignment was the Hailstone series problem, the third was to play games of "rock, paper, scissors," and the last required loading and processing of arrays.

Students were assigned to one of two conditions according to their preferences: the experimental condition with subjects using the CPP environment over a network with a human tutor, and a control condition allowing students to go about writing their programs as they normally would (without CPP). The first project acted as our pre-test: all students did the assignment on their own with no interaction. For the experimental condition, subjects engaged in CPP to prepare for the middle two projects. No CPP interaction was provided in the final assignment, thus permitting it to play the role of post-test.

Data collection consisted of recording compiler and editor activities during implementation of all subjects. Addition-

ally, all files submitted to the compiler were saved (similar to [14]). This was done with the students' knowledge and consent.

To determine if CPP subjects displayed less erratic behavior during implementation, we borrow the idea of *floundering* from the fields of user interfaces and intelligent tutoring systems. For computer programming, we define floundering as repeated attempts to repair a bug leaving the program no closer to being correct after each attempt. We applied two measures to gauge floundering. The first was simply to count the number of successful compile attempts (i.e., no syntax errors) in the hope that students who floundered more spent more time compiling and running their programs. The second was to look at the content of the differences between the syntactically correct versions of their programs. We categorized the changes between each pair of files as involving an *algorithmic bug-fix* or something else.[2] We then counted the number floundering *episodes* involving algorithmic bugs. An episode is defined as one more recompiles intended to fix the same error. Intuitively, the number of floundering episodes found in an implementation represents the number of algorithmic impasses the student struggles to repair.

To see if students who used CPP were more comfortable using natural language in their programs, we simply counted the number of comment lines in the final version of the post-test program. Single line comments counted as 1, as did each line within multi-line comments. Administrative comments (name, date, class, etc.) were not counted.

Lastly, to determine if CPP subjects gained a better understanding of structural concepts, we examined indentation in all files submitted to the compiler, including those with syntax errors. Each improperly indented line was counted upon its first appearance. Our rules for proper indentation were liberal (one space was considered enough), but consistent with those presented in popular introductory programming textbooks.

4.3 Results

Because of a possible self-selection bias (subjects were assigned to one of the two conditions according to their stated preferences), the first project included in the study, also the first non-trivial assignment given in the class, was used as a pre-test. The scores on these projects revealed no significant correlation between condition and pre-test ($t(12) = -0.269$, $p = 0.79$), allowing us to conclude that both conditions consisted of subjects of equivalent competence.

Floundering The first test for floundering was to simply count all successful compile attempts. CPP subjects compiled an average of 25.0 ($sd = 32.9$) times during their post-test project implementation, and control subjects compiled 49.4 ($sd = 41.38$) times. The difference was not statistically significant. However, by throwing out a statistical outlier in the CPP condition who was 2.2 standard deviations from the mean, the difference was marginally significant ($F(1, 10) = 4.08$, $p = 0.071$).

The second test was to count episodes of algorithmic floundering. In the post-test assignment, CPP subjects had a mean of 1.00 ($sd = 2.24$) episodes and the control subjects averaged 2.71 ($sd = 2.63$). The difference is marginally statistically significant ($F(1, 11) = 3.53$, $p = 0.0872$) and moderately large (effect size = 0.65).

Because the identification of floundering is somewhat subjective, the implementation logs were coded independently by two experienced programming instructors. Using a specialized coding tool that displayed subsequent versions of the programs, each compile attempt was classified as an algorithmic flounder or not. After identifying the beginning and ends of all floundering episodes, the intercoder reliability was computed with a +/- 1 cushion on the boundaries for episodes of 2 in length or longer. In this study, four randomly chosen implementations were used for training and the rest to measure agreement. The result was a kappa value of 0.872.[3]

Commenting In their final post-test programs, CPP subjects had a mean of 35.0 ($sd = 23.1$) comment lines. The mean for control subjects was 12.3 ($sd = 9.76$), leaving a difference of nearly 23 more lines of comment lines on average for CPP subjects over control subjects. This difference is statistically significant ($F(1, 11) = 5.97$, $p = 0.0325$) and large (effect size = 2.33).

Indentation Because indentation had been covered in class and gradually learned throughout the semester, there was a ceiling effect when analyzing the indentation of the post-test program. Thus, we report results for the second project understanding that the CPP students had already created a pseudocode solution prior to their real solution.

During the implementation of the second project, CPP subjects produced an average of 2.43 ($sd = 2.88$) improperly indented lines of code per implementation. The mean for the control subjects was 12.7 ($sd = 10.8$). This means that CPP subjects maintained the structure of their code by incorrectly indenting roughly 10 less lines than the control subjects. This difference is statistically significant ($F(1, 11) = 5.61$, $p = 0.0373$) and large (effect size = 0.95).

5 Discussion

Overall, we feel these results justify a dialogue-based approach to novice program design. If difficulty at programming is a reason for novices to drop out of introductory courses, it might be that extra support early in the programming process, such as that provided by CPP, might avert many of the frustrations they encounter. It also seems that students receiving CPP adopt more desirable behaviors during implementation. By helping students build pseudocode first, they are receiving a certified correct outline of a program that they are responsible for creating. They still face the unavoidable struggles of learning a specific programming language, but can do so without as much confusion regarding the particular algorithms they are trying to implement.

Regarding comments, we feel that these results support our argument that coached program planning encourages students to think about solutions in their own words, and to feel more comfortable inserting comments into their code. The mere existence of comments is only a suggestion, however. An analysis of the content of these comments may be necessary to better test our hypothesis.

[2]These include things like tweaking i/o behavior, adding comments, superficial rearrangement of program code, and adding large code segments.

[3]Kappa is a popular measure for intercoder reliability because it factors out agreement by chance. Generally, a kappa value above 0.80 is considered reliable.

As stated above, CPP students saw the proper indentation in the pseudocode in the second project. The results support the claim that CPP students produced fewer incorrectly indented lines in their actual code. Since we could not use the final project for this test, the result is not as strong regarding the longer term effect of CPP. For indentation, this is perhaps not a serious criticism.

The results involving floundering were not as conclusive as we had hoped, but still suggestive that students who are trained to think about pseudocode and their algorithm in a larger sense will spend less time floundering during implementation. There are several possible reasons for the inconclusive gains with the floundering measure. Students were only trained on two assignments – at about an hour per tutoring session, this amounted to roughly two hours of training per student in the experimental group. Thus, perhaps more programs with CPP would have helped. It is also possible that individual differences and tendencies were too great suggesting that more subjects might be necessary for this particular measure. This was certainly a problem given the cost of using a human tutor, but will not be when we repeat the evaluation with an ITS.

6 Future Work

We are building a dialogue-based intelligent tutoring system called the *Pseudocode Tutor* to perform CPP. Currently, the system rigidly follows the 4-step pattern found in the dialogues and uses keywords to understand student input. Advanced understanding and tutoring strategies are being developed, but not part of the tutor as of this writing. Our goal is to have the ITS behave as closely as possible to the human tutor in this study. At the very least, we hope to simulate many of the effective tutoring strategies present in our corpus (e.g., referring to an example, posing completion and Socratic-style questions, simulating execution of the pseudocode, etc.). As noted earlier, free-form pseudocode presents difficult natural language understanding challenges that still lack general solutions. To skirt this problem somewhat, the Pseudocode Tutor is being built to follow the standard presented by Robertson [12] which requires explicit references to variable names, limited use of reserved words, and some restrictions on the format of steps. The overall feel is nonetheless intuitive and natural.

Regarding the analysis of the data, we are also interested in analyzing various aspects of the student projects, including the quality of the designs, quality of the identifiers, content and quality of the comments, the timing of when comments are added, and overall attitude regarding their work. We believe that CPP should have an impact on these aspects of novice programming in positive ways.

While we have focused on structured programming exclusively for now, we believe the idea of using natural language to prepare a student to write real code would also be effective in other paradigms. For example, in their book *How to Design Programs*, Felleisen et. al. present design recipes to help students write functions in Scheme [5]. Each phase in these recipes has the student draw on intuition and use natural language to guide code writing. When students ask for help, most tutorial interaction involves asking general questions about the student's status within the design recipe [6]. The role of natural language in novice programming is clearly an important avenue that deserves continued attention.

7 Acknowledgements

This research was supported by NSF grant number 9720359 to CIRCLE, the Center for Interdisciplinary Research on Constructive Learning Environments at the University of Pittsburgh and Carnegie-Mellon University. We would also like to thank Mark Fenner for his help with the coding work, and Bob Hausmann for assistance in the data analysis.

References

[1] Bonar, J. G., and Cunningham, R. Bridge: Tutoring the Programming Process. In *Intelligent Tutoring Systems: Lessons Learned*, J. Psotka, L. D. Massey, and S. A. Mutter, Eds. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1988, pp. 409–434.

[2] Bruckman, A., and Edwards, E. Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. In *Proceedings of the Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), pp. 207–214.

[3] Chi, M., Bassock, M., Lewis, M., Reimann, P., and Glaser, R. Eliciting Self-explanations Improves Understanding. *Cognitive Science 18* (1994), 439–477.

[4] DuBoulay, B. Some Difficulties of Learning to Program. *Journal of Educational Computing Research 2*, 1 (1986), 57–73.

[5] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. *How to Design Programs*. MIT Press, 2001.

[6] Flatt, M. Personal Communication, March 2002. SIGCSE02 Dr. Scheme Workshop.

[7] Joni, S.-N., and Soloway, E. But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research 2*, 1 (1986), 95–125.

[8] Lee, P., and Phillips, C. Programming Versus Design: Teaching First Year Students. *SIGCSE Bulliten 30*, 3 (1998), 289.

[9] Pane, J. F., Ratanamahatana, C. A., and Myers, B. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54* (2001), 237–264.

[10] Pintrich, P. R., Berger, C. F., and Stemmer, P. M. Students' Programming Behavior in a Pascal Course. *Journal of Research in Science Teaching 24*, 5 (1987), 451–466.

[11] Rist, R. S. Schema Creation in Programming. *Cognitive Science 13* (1989), 389–414.

[12] Robertson, L. A. *Simple Program Design*. Course-Technology – Thompson Learning, 2000.

[13] Shackelford, R. *Introduction to Computing and Algorithms*. Addison-Wesley, 1998.

[14] Spohrer, J. C., and Soloway, E. Putting It All Together is Hard For Novice Programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (Tucson, Arizona, November 12-15 1985).