The Effects of Emphasizing Computational Thinking in an Introductory Programming Course

Stephen Davies
University of Mary Washington, stephen@umw.edu

Abstract - **In many introductory programming courses, the surface features of the programming language can distract and intimidate students so much that they fail to concentrate on what is really the "brainy" task: solving the problem conceptually. To counter this, we devised a form of structured pseudocode, designed to highlight and facilitate algorithmic construction so that the complexities of the programming language can be deferred until proficiency in design has been reached. Students taught with this experimental approach are not introduced to the language itself, or the compiler, until the last few weeks of the semester. A controlled experiment comparing this approach with a traditional language-based pedagogy has revealed that by the end of the course, students' programming skills, even on language-specific tasks, is every bit as strong as students taught traditionally, and that their comfort level with modularity (writing functions) is increased. Additionally, we found that students appear to strongly prefer such an approach, citing mostly emotive benefits, and that these effects may be particularly strong among women.**

Index Terms – Computational Thinking, Pedagogy, Pseudocode

INTRODUCTION

Why is programming such a difficult skill to acquire? We contend that it is partially because novices are forced to learn too many different things at once. In particular, newcomers are required to simultaneously tackle the following two very formidable tasks:

1) learning how to combine computational building blocks (including data storage, branching, repetition, modularity, etc.) into a correct solution, and
2) expressing this assembled solution in a strange and unforgiving language unlike anything they've ever previously encountered.

These correspond to the "notional machine" and "formal language notation" areas in DuBoulay's taxonomy of difficulties in learning to program[1], and each brings with it considerable difficulties.

As educators, we all realize that the first of these tasks is where the "meat" of the problem lies. Programming is the art of structured problem-solving, of algorithmic expression, an eminently creative endeavor that demands elegance and precision. Yet we have noticed that for a majority of beginners, the second task is so alien and intimidating that it dominates the student's attention and prevents them from focusing on what is the most important – and should be the most rewarding – aspect of programming. After all, the most immediately arresting feature of their textbook and lecture notes is the programming language itself: a frightening world of semicolons and curly braces whose secret meaning takes great pains to discover. Students are often so distracted by these surface features in languages like C++ that they fail to concentrate on what is really the "brainy" task: solving the problem conceptually.

We believe it may be wise to split up these two tasks as much as possible. First, get students used to the idea of what programming fundamentally *is*: combining algorithmic constructs to solve problems methodically. Then, once this paradigm has been understood, introduce the programming language as a way of expressing these algorithms for a computer to run. This approach is akin to first teaching a high schooler how to structure and compose a term paper, and then separately instructing them in a foreign language, rather than trying to do both at once and expecting a high-quality 5-page Spanish essay to result. These are separate skills, with separate challenges, and each needs the full attention of the student to grasp. Our experience has shown that students find languages like C++ daunting, and struggle to write correct programs with it. But if equipped first with firm ideas (and practice) in *how* to structure a program, we predict that facing the language itself would be less of a blow, since students would already have an idea of what to do with it.

This paper describes our initial experimentation with a strongly pseudocode-based pedagogy, the idea being to delay introduction of the programming language itself so that these two facets of developing software can be treated separately. A closely related goal is to sidestep the "programming phobia" novices often experience when seeing a language like C++ for the first time by letting them gain confidence with problem-solving first.

RELATED WORK

Our approach is not the first to use pseudocode, obviously. Many instructors briefly introduce the technique in an introductory programming course before diving into language constructs. Textbooks such as [2] and [3] make use of it, admonishing students to "Develop a general solution first," "keep your thoughts straight," and "think first and code later!"[2, p.5]. But usually little time is spent on

978-1-4244-1970-8/08/$25.00 ©2008 IEEE

actually developing sophisticated pseudocode solutions. The aim is primarily to warn students not to rush to the keyboard too quickly, before thinking through the problem.

Various attempts to expand the use of pseudocode beyond this are mentioned in the literature, for instance Olsen's work[4], in which approximately 25% of lecture time was devoted to working with natural-language pseudocode. This was an effort to remedy the "lack of problem-solving skills" among computer science majors that her institution identified, and to thus better prepare students for the upper end of the curriculum.

Our work differs from approaches like Olsen's primarily in two ways. First, we spent far more time on pseudocode during the semester than she (approximately 66% of the course instead of 25%), covering pseudocode and programming language strictly in parallel, rather than interleaving them. Second, and most importantly, our students' pseudocode, though made up of natural language, is highly restricted to a set of predefined constructs. It is not merely an informal narrative description of a solution.

The idea of actually composing formal or near-formal solutions in natural language has always been compelling; it motivated the English-like syntax of languages like COBOL[5], for instance. Experiments with children have confirmed that natural language programming is indeed easy for newcomers to grasp, and that the errors they make rarely arise from issues with the natural language aspect.[6] A recent report [7] describes the innate skills in algorithmic development that most beginners seem to bring with them, and suggests that leveraging their ability to write natural language solutions could be used effectively in programming. All this seems to confirm a major premise: that most students would find it more natural to learn how to write programs in structured pseudocode.

This is not to suggest that natural language alone is a panacea. In many cases the deeper structure of a typical novice's solution veers sharply away from the constructs available in popular programming languages, regardless of whether the surface syntax is in natural language.[8] Regardless, presuming that an instructor does enforce the underlying structure of an imperative programming language, it would seem that expressing this structure in natural language would be more palatable for beginners than a C++-like syntax.

Note that although in our approach the pseudocode solutions are "language-independent," they are nevertheless restricted to imperative programming languages. Ours is not an attempt to construct a paradigm-general pseudocode, as in [9]; rather, we are intentionally steering students towards competency with the languages most commonly in use today, and most common in introductory courses.

THE STRATEGY TOOLBOX

Our experimental approach was to first introduce students to the concept of program *strategies*. A *strategy* is a block of structured pseudocode describing how to solve a problem algorithmically. It differs from traditional pseudocode in that very little flexibility is given to the student to deviate outside certain prescribed elements. We devised a "strategy toolbox" full of a small number of English-language building blocks that could be combined to make strategies, as outlined in figure 1. We were quite insistent that students' strategies be comprised only of these strategy tools, exactly as shown. (Certain stylistic deviations, such as writing "...follow these steps:" instead of "...do these steps:" were permitted but not encouraged.)

Information holders: *(variables; always in all-caps)*
#. Make SOMETHING hold (some value or basic computation).
#. Remember what the user says, and call it SOMETHING.
Input/Output:
#. Ask the user, "..."
#. Tell the user, "..." *(may include information holders)*
Forks in the road: *(branching)*
#. If (something is true), do these steps:
  a. ...step 1...
(optional) Otherwise, do these steps:
  a. ...step 1...
Repetition: *(loops)*
#. If (something is true), do these steps, *then check it again:*
  a. ...step 1...
Interchangeable parts: *(functions; name always underlined)*
PartName
1. Call Input #1 SOMETHING.
  ...(body, composed of strategy tools)...
#. Stop. The output is (some value).

To use *(or "call")* an interchangeable part:
#. Make SOMETHING hold the output of PartName where Input #1 is (some value), Input #2 is (some value), ...

FIGURE 1
THE PRINCIPAL TOOLS IN THE STRATEGY TOOLBOX AVAILABLE FOR STUDENTS TO USE IN SOLVING PROBLEMS.

A finished strategy (e.g., figure 2) is thus a natural-language expression of an algorithm to solve the problem. Unlike in traditional pseudocode, however, there is very little guesswork in interpretation here. A strategy is not a free-form English description, featuring widely varying forms of expression and inconsistent degrees of precision. It is, in fact, constrained in such a way that it is ultimately translatable, line by line, into any imperative programming language. More importantly, we claim that *a correct strategy completely and properly solves the problem*, which is of course the student's main goal. After the strategy has been written, the only things that remain are language-specific implementation issues, nothing fundamental to the problem itself.

The metaphor used to motivate strategies was that of the "human computer." We discussed how the term computer originally designated "a human who performed computation," rather than a machine. Such human computers typically did not actually understand anything about the meaning of the scientific or business calculations they carried out: they were mere slaves, faithfully executing "what" without asking "why," in exchange for some small wage. A strategy, then, is a sort of contract between the two parties. And since the worker cannot be trusted with any

knowledge or reliable interpretation, the only hope for a correct answer is to specify the instructions so precisely and unambiguously that they are sure to be carried out without mistake. The toolbox, students were told, contained exactly the set of tools that they could count on being executed correctly.

At this point one might object, "if strategies must be specified in such step-by-step detail, why use them at all? Why not present C++ (or another language) to begin with?" The answer is twofold. First, the resulting artifacts are in easy English, understandable by even the uninitiated. This is clearly easier for many novices to work with[6], and much less frightening than the peculiar symbols, abbreviated keywords, stringent capitalization rules, and odd parenthesization of most modern programming languages. (Consider the two halves of figure 2 from the perspective of someone who has never programmed before.) Second, even when the essential algorithmic steps are spelled out to this degree, there are still numerous less fundamental language-specific issues that can be postponed instead of overloading the student's mind with them prematurely. Even in the sample strategy in figure 2, we have safely avoided dealing with variable types (a concept foreign to most novices' thinking[7]), include files, namespaces, declarations, function prototypes, casting, C++ parameter-passing syntax, and decimal truncation due to integer division.

Space does not permit explanation for all of our choices in designing the strategy toolbox. The specifics of the strategy language are not most important anyway: many alternate syntaxes would work as well. The crucial points are: (1) a written strategy must be very clearly understandable, even at first glance, (2) the tools encompass all constructs necessary for procedural programming, and (3) each tool must be relatively easily converted into an imperative programming language when this becomes desirable later on.

We will comment briefly on two features we settled on after some trial and error. First, the "outline numbering" of the steps in a strategy proved immensely helpful. At first, we tried to specify control flow simply through indentation (as in Python), but found this was very unnatural for students and led to many misunderstandings. It turned out to be much more effective to force students to explicitly specify the sequence of steps, and (importantly) the ends of conditional and loop blocks, by numbering their strategy steps, and then demonstrating how this corresponded to C++ curly braces.

Second, a word about "interchangeable parts" (the strategy equivalent of C++ functions/subroutines.)

Start here:
1. Make PLAYER NUM hold 1.
2. Ask the user, "How many at bats for #PLAYER NUM?"
3. Remember what the user says, and call it AT BATS.
4. Make OUTF AT BATS hold 0.
5. Make OUTF HITS hold 0.
6. If AT BATS is not -1, do these steps, *then check it again*:
   a. Ask the user, "How many hits for player #PLAYER NUM?"
   b. Remember what the user says, and call it HITS.
   c. Ask the user, "What is player #PLAYER NUM's position?"
   d. Remember what the user says, and call it POSITION.
   e. If POSITION is either "leftfield," "centerfield," or "rightfield," do these steps:
      i. Make OUTF AT BATS hold OUTF AT BATS plus AT BATS.
      ii. Make OUTF HITS hold OUTF HITS plus HITS.
   f. Make PLAYER NUM hold PLAYER NUM plus 1.
   g. Ask the user, "How many at bats for player #PLAYER NUM?"
   h. Remember what the user says, and call it AT BATS.
7. Make BATTING AVG hold the output of Batting Average where input #1 is OUTF AT BATS and input #2 is OUTF HITS.
8. Tell the user, "The outfielders batted BATTING AVG this game."

Batting Average: (interchangeable part)
1. Call input #1 TOTAL AT BATS.
2. Call input #2 TOTAL HITS.
3. Make THE BATTING AVERAGE hold TOTAL HITS divided by TOTAL AT BATS.
4. Stop. The output is THE BATTING AVERAGE.

```
#include<iostream>
#include<string>
using namespace std;

float battingAverage(int totalAtBats, int totalHits);

main() {
  int playerNum, atBats, hits, outfAtBats, outfHits;
  string position;
  float battingAvg;

  playerNum = 1;
  cout << "How many at bats for player #" << playerNum << "?";
  cin >> atBats;
  outfAtBats = 0;
  outfHits = 0;
  while (atBats != -1) {
    cout << "How many hits for player #" << playerNum << "?";
    cin >> hits;
    cout << "What is player #" << playerNum << "'s position? ";
    cin >> position;
     if (position == "leftfield" || position == "centerfield" ||
         position == "rightfield") {
          outfAtBats += atBats;
          outfHits += hits;
   }
    playerNum++;
    cout << "How many at bats for player #" << playerNum << "?";
    cin >> atBats;
  }
  battingAvg = battingAverage(outfAtBats, outfHits);

  cout << "The outfielders batted " << battingAvg <<
    "this game. " << endl;
}
float battingAverage(int totalAtBats, int totalHits) {
  float theBattingAverage;
  theBattingAverage = ((float) totalHits) / ((float) totalAtBats);

  return theBattingAverage;
}
```

FIGURE 2
A SIMPLE "STRATEGY," AND EQUIVALENT C++ PROGRAM

978-1-4244-1970-8/08/$25.00 ©2008 IEEE

Originally we had mixed feelings about introducing the concept of modularity into the toolbox, feeling that perhaps this was best left to the language-specific part of the course. We discovered, however, that nothing could be further from the truth. Of all the successes the strategy approach offered, none was greater than in helping students understand how (and why) to call (and write) functions. Though we did not think to test for this explicitly, there was nevertheless a noticeable difference in students' willingness to break out common tasks into interchangeable parts when compared with C++ functions. Perhaps this is because the syntax of C++ is so daunting in this regard, and dampens students' enthusiasm to complete the work of setting up function prototype, typed and named parameters, return value, return statements, and the other necessary mechanics that come along with creating a function in C++.

Note also that we chose to use numbered, positional arguments for parameter-passing. This turned out to be close enough to the way C++ passes parameters to make the strategy-to-C++ translation easy, and the explicit numbering ("Call Input #1 DISTANCE," etc) turned out to aid student understanding. This surprised us; we had expected that the extra syntactic baggage of specifying a number for each input would merely add clutter to the page and to students' minds. But the difficulty in mapping actual parameters to formal parameters seems to be greatly reduced when explicit tags (in this case, numeric) are attached to each, accentuating the correspondence.

STRATEGY WORKSHEETS

In analyzing and writing strategies, we emphasized the use of a "strategy worksheet." This played the role of the "human computer's" personal notepad, on which he/she kept track of the values of information holders (variables) as he/she blindly carried out the strategy in question. We did not stress this technique at first, assuming that it would be relatively easy for students to keep a small number of values in their minds as they stepped through the strategy. But this false expectation was undoubtedly because as instructors, our relative level of experience with programming has made this process second nature. Without a worksheet, we found that students often failed to appreciate the importance of what data the information holders actually held during the course of the program. They tended to look solely at the strategy, without thinking of the changing state of the variables over time. Forcing them to maintain a worksheet during computation, and to cross out and subsitute new values explicitly every time the strategy called for it, was immensely helpful in bringing this point home.

CONTROLLED EXPERIMENT

To determine the strengths and weaknesses of the approach, we carried out a controlled experiment in two sections of CPSC 110, an introductory programming course which satisfies a general education requirement for all students and which serves as the prerequisite for the computer science major sequence. The same instructor taught both sections, using the same examples, the same homework problems, and the same selection of material:

Section 1 (control group): from the beginning, C++ was introduced. After some very brief remarks on algorithms in the first two lectures, students learned how to work with the compiler in a Unix environment, and focused on writing and debugging actual, running programs. (26 students.)

Section 2 (experimental group): developing strategies for solving problems was the sole focus for the first ten weeks of the course. Students used the strategy toolbox to write and analyze non-executable solutions. Only in the last third of the semester was C++ introduced, and students were then taught how to "translate" their strategies into C++ to compile and run them. (22 students.)

The purpose of this experiment was to discover the overall effect of teaching almost an entire semester of introductory programming without reference to a programming language. Would students attain the required competency? Would concentrating so much on algorithmic development (as opposed to syntax) result in programmers better able to solve problems conceptually? Would students enjoy the new approach, or regard it as a nuisance?

I. Results: skills

At the end of the semester, students from both groups participated in a number of graded activities designed to test various aspects of their programming abilities. At this time, students from section 1 had experienced about thirteen weeks of instruction and practice in C++ programming; students from section 2 had experienced about ten weeks of algorithmic development using the strategy toolbox, and only three weeks of C++ practice.

The results are summarized in Table I. As can be seen, none of the activities revealed any obvious difference between the two groups' skills. We used a statistical difference of means test to analyze the two groups, in which a student's-t statistic of at least 1.33 (representing 90% confidence) was required to demonstrate significance.

TABLE I
SUMMARY OF SKILLS TEST RESULTS FOR THE TWO GROUPS. THE FAR-RIGHT COLUMN CONTAINS STUDENT'S-T VALUES WHICH MUST REACH 1.33 TO INDICATE STATISTICAL SIGNIFICANCE.

Activity	*Section 1*	*Section 2*	*t*
Timed C++ programming	36.0/50	34.2/50	.91
Algorithmic description	42.0/50	41.3/50	.47
Detecting C++ program errors	11.3/15	12.1/15	1.31
Adaptability to other languages	40.8/50	43.7/50	1.17

Timed C++ programming: students were given 45 minutes in a laboratory setting to write, compile, and debug three short C++ programs of varying difficulty. (For instance, one program was to prompt the user for an arbitrary-length sequence of positive integers, terminated by a "-1" input, and then print out the two largest values in the

sequence.) The problems were chosen deliberately to put time pressure on the students; it was expected that only a few would have time to solve all three correctly.

There was no statistically significant difference between the two groups, indicating that students taught with strategies do not have significantly more difficulty writing C++ programs spontaneously.

Algorithmic description: students were given 45 minutes in a classroom setting to write in English sentences (*not* in "strategy language" unless section 2 chose to do so) a description of how they would go about solving five different problems. Most of these involved examples from the "Yahtzee" dice game; e.g., if the user inputs five integers in the range 1-6, how would you determine whether the roll constituted a Yahtzee (5-of-a-kind)? A 3-of-a-kind? A full house? Etc. A typical solution was a paragraph of six to ten sentences. Points were awarded for correctness and precision of description; glossing over the real difficulties of the problem resulted in very little credit.

Again, neither group outperformed the other significantly. We had expected that perhaps the experimental group would do better here, since they had spent more time on algorithms proper, but this difference did not materialize.

Detecting C++ program errors: students were given paper copies of a one-page C++ program that contained fifteen common mistakes. These included both syntax errors and common C++ run-time problems (e.g., using "=" instead of "==" for comparison; incorrectly assuming that the value of an uninitialized variable would be set to 0.) Participants had to identify and describe, but not correct, each mistake.

Remarkably, the experimental group outscored the control group on this activity, and to a degree just reaching statistical significance (90% confidence.) We expected that of all the activities, the experimental group would be at the greatest disadvantage here, since the test involved language-specific knowledge. Perhaps the recency of the exposure to the material helped the experimental group.

Adaptability to other languages: we invented our own hypothetical language (called "S++") whose surface features looked radically different than C++, but whose underlying structure was the same. We then presented students with an exercise in which they were given a complete S++ program, and told what it did. After studying this program, they were asked to predict the output of another S++ program and try and write their own. We predicted that the experimental group would have an advantage here, since they were used to thinking in terms of language-independent algorithms. They did outscore the control group, but not to a statistically significant degree.

Overall, these results do not demonstrate any direct benefit of a pseudocode-based pedagogy on students' programming skills, or even algorithmic thinking. They do, however, allay a possible fear: that fluency with the specifics of the programming language might suffer as a result of less exposure. This does *not* appear to be the case.

II. Results: attitudes

Students completed anonymous questionnaires at the end of the semester designed to gauge their attitudes on a number of issues. One set of questions was given to all students, and a second set only to those taught with the experimental method. Unless indicated otherwise, all items were statements with which students were asked to agree or disagree on a 5-point Likert scale (1 means "strongly disagree" and 5 "strongly agree.")

General attitudes. Most of the questions about general attitudes to programming revealed no significant difference between the two groups. For example, responses to items like "Writing C++ programs is frustrating" and "I think with enough practice, I would help write a large, cool application like Microsoft Word or Facebook" were essentially the same.

One of the few differences was on the item "For me, the most difficult thing about writing a computer program is coming up with the basic algorithm to solve the problem." The control group averaged 2.81 on this item, and the experimental group 3.81 (for a very significant student's-t statistic of 3.02.) This could be interpreted in a number of ways. We believe that it supports our major hypothesis. To our way of thinking, students taught with the strategy method are more likely to (correctly) believe that the crux of the problem is creating the algorithm. By contrast, students taught with a traditional pedagogy are more often "fooled" into thinking that the language is the major source of complexity, and the algorithm secondary.

Two gender-specific items were particularly interesting. First, students taught with the experimental approach were significantly less likely to agree with the statement, "Men are usually better at computer programming than women are" (experimental average: 1.81, control average: 2.27, for a student's-t of 1.42.) Second, when considering women *only*, students from the experimental group were slightly more likely to agree with "Writing computer programs involves a lot of creativity" (experimental: 4.07, control: 3.56, student's-t: 1.24.) Interestingly, no such difference was found among male students between the two groups. In any event, it appears that this approach may have a modest effect on increasing confidence and interest in programming among women.

Perceived difficulty of algorithmic constructs. We wanted to find out how hard each group found it to use the various programming building blocks (variables, loops, branches, etc.) to see whether the strategy method made any of them easier. Each group was asked a set of questions on a 1-5 scale, where 1 meant "very hard to use" and 5 "very easy to use." For the control group, the items were "variables," "if," "while," "functions," etc., and for the experimental group, the corresponding strategy tools "information holders", "if...do these steps," "if...do these steps, then check it again," "interchangeable parts," etc.

We found no significant difference between the two groups for any of the tools *except* functions, where there was a noticeable difference. The strategy group found using

and creating interchangeable parts to be a 3.82 on the "easiness" scale, while the control group averaged only a 2.85 for calling and writing functions. (The student's-t is 3.2.) This corresponds to our intuition from in-class experiences, where students from the strategy group seemed to grasp the concept of modularity more quickly. Too, we sometimes perceived students from the strategy group to be more willing to spontaneously write their own functions when solving problems. The data here is sparse, but a tentative conclusion is that the C++ function calling syntax is chiefly responsible for students' inhibitions in creating functions, and that eliminating this syntactic obstacle frees up modular thinking.

TABLE II
AVERAGE RESPONSES TO ATTITUDE ITEMS
(1 = "STRONGLY DISAGREE", 5 = "STRONGLY AGREE.")

"The strategy toolbox was a pointless exercise that distracted from the main task of learning how to program a computer." (1.63)
"The strategy toolbox was a helpful way of learning how to think about constructing an algorithm without being encumbered with programming language syntax." (4.38)
"I was bored learning strategies because I knew that the computer couldn't actually execute them." (1.95)
"Concentrating first on strategies made it easier for me to learn how to write computer programs." (4.2)
"It would have been just as easy to learn how to write a computer program by studying C++ from the beginning, instead of first focusing on devising strategies to solve problems." (2.27)
"I predict that in future programs I may write, I will think about the strategy toolbox in deciding how to construct them." (3.95)
"I would have been intimidated to learn C++ directly without first learning how to use the strategy toolbox." (3.85)

Reaction to the strategy approach. Finally, the most revealing aspect of the surveys, and the one the recommends the strategy approach the most highly, is simply that students seem to prefer it. We were frankly stunned by the unanimity of the written comments we received. We had hoped that perhaps half the students would appreciate this "softer" introduction to programming, while the other half, eager to dive into programming proper, would view it as a waste of time. By contrast, only one written survey had an overall negative judgment, and even this criticism was restrained: "I think the strategy toolbox was a good idea that just didn't work for me personally." The vast majority of students were effusive in their praise: "I thought C++ was so much easier after doing strategies!" "I thought the strategy toolbox was very helpful and wouldn't change anything about it." "Everything was extremely helpful!" "I am really happy that I learned strategy toolbox first, if not, I don't think I will able (sic) to write C++ code." Etc. We also included a few specific "reaction" items on the survey, to get a quantitative measure. The average responses from the experimental group (again on a 5-point Likert scale) for select items are presented in Table II.The last item is particularly interesting, since it speaks to the emotional response many students have when confronted with unfamiliar notation. Whether or not the strategy method actually made them better programmers, many students *perceive* that it helped them. And given declining enrollments, it seems that anything we can do to curb the intimidation factor would be welcome.

CONCLUSIONS

Learning the general techniques of programming first without the accompanying complexities of an actual programming language does not appear to result in greater competency overall. But the approach does no worse than the traditional pedagogy, even in areas one might suspect it would (e.g., recognizing programming language errors.) It also appears to facilitate modular thinking by diminishing the barriers that novices often face in formulating subroutines properly in a programming language. Interestingly, too, modest evidence suggests that women may especially benefit from the approach, particularly in terms of their confidence and attitudes about programming. Given these benefits, and the fact that most of our students enthusiastically lauded the approach, a pseudocode-based pedagogy may be very advantageous in "softening the blow" for novice programmers.

REFERENCES

[1] DuBoulay, B. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), pp.57-73, 1986.

[2] Dale, N.B. and Weems, C. *Programming in C++*. Jones & Bartlett Publishers, 2004.

[3] Malik, D.S. *C++ Programming: From Problem Analysis to Program Design*, 3rd Edition. Course Technology, 2006.

[4] Olsen, A.L. Using pseudocode to teach problem solving. *Journal of Computing Sciences in Colleges*, 21(2), pp.231-236, 2005.

[5] Sammet, J.E. The early history of COBOL. In R. Wexelblat, Ed. *History of Programming Languages.* New York: Academic Press, 1981.

[6] Bruckman, A. and Edwards, E. Should we leverage natural-language knowledge? *Proceedings of SIGCHI 1999*, pp.207-214, 1999.

[7] Chen, T.Y. et al. Commonsense computing: using student sorting abilities to improve instruction. *ACM SIGCSE Bulletin* 39(1), 2007.

[8] Pane, J.F., et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54, pp.237-264, 2001.

[9] Wells, M.B. and Kurtz, B.L. Teaching multiple programming paradigms: a proposal for a paradigm general pseudocode. *Proceedings of the 20th SIGCSE symposium*, 1989.

[10] Wing, J.M., "Computational Thinking*," Communications of the ACM*, 49(3), 2006, pp.33-35.