Progress Reports and Novices' Understanding of Program Code

Linda Mannila
Dept. of Information Technologies, Åbo Akademi University,
Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linda.mannila@abo.fi

ABSTRACT

This paper introduces *progress reports* and describes how these can be used as a tool to evaluate student learning and understanding during programming courses. A progress report includes a short piece of program code (on paper), covering topics recently introduced in the course, and four questions. The two first questions are of a "trace and explain" type, asking the students to describe the functionality of the program using their own words - both line by line and as a whole. The two other questions aim at gaining insight into the students' own opinions of their learning, as they are asked to write down what they think they have learned so far in the course and what they have experienced as most difficult.

Results from using progress reports in an introductory programming course at secondary level are presented. The responses to the "trace and explain" questions were categorized based on the level of overall understanding manifested. We also analyzed students' understanding of individual programming concepts, both based on their code explanations and on their own opinions on what they had experienced as difficult. Our initial experience from using the progress reports is positive, as we feel that they provide valuable information during the course, which most likely would remain uncovered otherwise.

1. INTRODUCTION

In [5], we reported on a study from teaching introductory programming at high school level.[1] The results showed that abstract topics such as algorithms, subroutines, exception handling and documentation were considered most difficult, whereas variables and control structures were found rather straight forward. These results were well in line with those of other researchers (e.g. [6, 7]). In addition, our findings indicated that most novices found it difficult to point out their weaknesses. Moreover, exam questions asking the students to read and trace code showed a serious lack in program comprehension skills among the students. One year later (2005/2006) we conducted a new study, in which we further investigated these issues using what we have called *progress reports*. This paper presents the results from this study.

We begin the paper with a background section, followed by a section describing the study and the methods used. Next, we present and discuss the results, after which we conclude the paper with some final words and ideas for future work.

2. BACKGROUND

Introductory programming courses tend to have a strong focus on construction, with the overall goal being students getting down to writing programs using some high-level language as quickly as possible. This is understandable, since the aim of those courses is to learn programming, which is commonly translated into the competence of using language constructs. Being able to write programs is, however, only one part of programming skills; the ability to read and understand code is also essential, particularly since much of a programmer's time is spent on maintaining code written by somebody else. In this paper, we refer to reading a program as "the process of taking source code and, in some way, coming to 'understand' it" (p. 1), as defined by Deimel [3].

One could assume that students learning to write programs automatically also learn how to read and trace code. Research has, however, indicated that this is not always the case. For instance, Winslow [15] notes that "[s]tudies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies" (p. 21). In 2004, a working group at the ITiCSE conference [8] tested students from seven countries on their ability to read, trace and understand short pieces of code. The results showed that many students were weak at these tasks.

Why then is it difficult to read code? Spinellis [13] makes the following analogy: "when we write code, we can follow many different implementation paths to arrive at our program's specification, gradually narrowing our alternatives, thereby narrowing our choices [...] On the other hand, when we read code, each different way that we interpret a statement or structure opens many new interpretations for the rest of the code, yet only one path along this tree is the correct interpretation" (p. 85-86).

Fix et. al [4] cite Letovsky, according to whom the overall goals of a program often can be inferred when reading the code, based on for instance variable names, comments and other documentation. Letovsky also suggests that the same thing applies to the data structures and actions of the program (i.e. the implementation): a person reading a program understands the actions of each line of code separately. The difficulty arises when trying to map high-level goals with their representation in the code. Lister et al. [9] talk about students failing to "see the forest for the trees" (p. 119). They have found that weak students, in particular, seem to have difficulties in abstracting the overall workings of a program from the code. This finding is supported by Pennington [10], who has found that whereas experts infer what a program does from the code, less experienced programmers make speculative guesses based on superficial information such as variable names, without confirming their guesses in any way.

Pennington [11] has also developed a model describing program comprehension, according to which a programmer constructs two mental models when reading programming code. First, the programmer develops a *program model*, which is a low-level ab-

[1] In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16-19. The main objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies.

straction based on control flow relationships (for instance loops or conditionals). This program-level representation is formed at an early stage and is inferred from the structure of the program text. After that, the programmer develops a *domain model*, which is a higher-level abstraction based on data flow, containing main functionality and the information needed to understand what the program truly does. Similarly, Corritore and Wiedenbeck [2] have found that novices tend to have concrete mental representations of programming code (operations and control flow), with only little domain-level knowledge (function and data flow).

In 1982, Biggs and Collis [1] introduced the SOLO (Structure of the Observed Learning Outcomes) taxonomy, which can be used to set learning objectives for particular stages of learning or to report on the learning outcomes. The taxonomy is a general theory which was not originally designed to be used in a programming context. Lister et al. [9] have used the taxonomy to describe how code is understood by novice programmers, and describe the five SOLO levels applied to novice programming as follows:

Prestructural ”In terms of reading and understanding a small piece of code, a student who gives a prestructural response is manifesting either a significant misconception of programming, or is using a preconception that is irrelevant to programming. For example, [...] a student who confused a position in an array and the contents of that position (i.e. a misconception)” (p. 119).

Unistructural ”[...] the student manifests a correct grasp of some but not all aspects of the problem. When a student has a partial understanding, the student makes [...] an ’educated guess”’ (p. 119).

Multistructural ”[...] the student manifests an understanding of all parts of the problem, but does not manifest an awareness of the relationships between these parts - the student fails to see the forest for the trees” (p. 119). For example, a student may hand execute code and arrive at a final value for a given variable, still not understanding what the code does.

Relational ”[...] the student integrates the parts of the problem into a coherent structure, and uses that structure to solve the task - the student sees the forest” (p. 119). For instance, after thoroughly examining the code, a student may infer what it does - with no need for hand execution. Lister et al. also note that many of the relational responses start out as multistructural, with the student hand tracing the code for a while, then understanding the idea, and writing down the answer without finishing the trace.

Extended Abstract At the highest SOLO level, ”the student response goes beyond the immediate problem to be solved, and links the problem to a broader context” (p. 120). For example, the student might comment on possible restrictions or prerequisites, which must be fulfilled for the code to work orderly.

Lister et al. found that a majority of students describe program code line by line, i.e. in a multistructural way. They argue that students who are not able to read and describe programming code relationally do not possess the skills needed to produce similar code on their own.

3. THE STUDY

3.1 Data

The data analyzed in this study were collected when two high school student groups (25 students) took an introductory programming course in 2005/2006. The majority of the students had no programming background. The courses were taught using Python and covered the basics of imperative programming. The data collection was conducted using surveys, progress reports and a final exam; in this paper we focus the analysis on the progress reports and on parts of the post course survey.

3.1.1 Progress Reports

A progress report can be seen as a type of learning diary (on paper), aiming at revealing the students’ own opinions and thoughts about their learning. What differentiates it from a traditional diary is that in addition to calling for self reflection, the report makes it possible for the teacher to evaluate students’ understanding based on their responses to ”trace and explain” questions. In this study each report included a piece of code, dealing with topics recently covered in the course, and four questions. First, in the ”trace and explain” questions, the students were to read the code and in their own words (1) describe what each line of the code does, and (2) explain what the program as a whole does on a given set of input data. In addition, students were asked to write down what they had learned and what had been most difficult so far in the course.

The first progress report was handed out after 1/3 and the second after 2/3 of the course. In total, we have analyzed 50 progress reports (two for each of the 25 students) and 25 post course surveys.

3.2 Method

The progress reports and post course surveys were analyzed manually in order to study three questions: how do students (1) understand program code as a whole, (2) understand individual constructs, and (3) perceive the difficulty level of different programming topics.

The first two questions were addressed by grouping the explanations given for the ”trace and explain” questions according to qualitative differences found in the data. On this point, the study resembles the SOLO study by Lister et. al [14] to some extent. However, in this study, we have explicitly asked students for both a multistructural and a relational response for each piece of code, giving us the possibility to compare how well the two responses match for individual students.

Finally, to address the third question, we studied the difficulty level of topics as perceived by the students by looking at both the progress reports and the post course survey. The ”trace and explain” questions also provided data pertinent to this part of the study as explanation errors were considered indications of student experiencing problems with that specific topic.

4. RESULTS AND DISCUSSION

4.1 Program Understanding

The ”trace and explain” code given in the first progress report is listed below (algorithm 1).

Students were asked to explain what this piece of code does if two integers are given as input. The analysis of the overall explanations gave rise to four categories:

Algorithm 1 Program given in progress report 1

```
try:
    a = input("Input number: ")
    b = input("Input another number: ")
    a = b

    if a == b:
        print "The if-part is executed..."

    else:
        print "The else-part is executed..."

except:
    print "You did not input a number!"
```

1. Correct explanation (n = 13)
2. Choosing the wrong branch in the selection after not explaining the meaning of the statement `a = b` (n = 5)
3. Choosing the wrong branch although having explicitly explained the statement `a = b` (n = 3)
4. Giving an output totally different from the ones possible based on the code (n = 4)

The first category is straight forward: over half of the students gave a perfect overall explanation for the program code, not only listing the output but also explaining the reasons for why that specific output was produced. The second category covers responses in which the student had failed to explain what the `a = b` statement means, and hence also missed the fact that when arriving at the selection statement, `a` does, in fact, equal `b`.

More concerning is the third category, in which students who have explicitly explained the `a = b` statement still believe that the else-branch will be executed. This indicates a misunderstanding related to either the results of an assignment statement, or the workings of the selection statement.

The fourth category appears to be some kind of guessing; the student thinks that the program outputs something that might have been expected from the code (e.g. values instead of one of the text messages), but was nevertheless incorrect.

Algorithm 2 Program given in progress report 2

```
def divisible(x):
    if x % 2 == 0:
        return True
    else:
        return False

default = 5
number = default

while number > 0:
    try:
        number = input ("Give me a number: ")
        result = divisible(number)
        print result
    except:
        print "Numbers only, please!"
```

The piece of code included in the second progress report is shown in algorithm 2. For this program, students were given a list of input data including positive integers and a character, ending with a negative integer. The explanations were analyzed in a similar manner to the corresponding task in the first progress report, and four categories were found:

1. Correct explanation (n=8)
2. Not understanding what it means to return a Boolean value (n=10)
3. Missing the last iteration, otherwise correct (n=4)
4. Incorrect output (n=3)

The number of correct overall explanations for the program in the second progress report was smaller than for the first report. We had expected that subroutines would be perceived as difficult, since these are commonly one of the main stumbling blocks in introductory programming [5, 6, 7]. However, the analysis revealed that subroutines per se were not necessarily the main problem; instead surprisingly many students had difficulties in understanding what happens when a subroutine returns a Boolean value. The code in algorithm 2 checks if the input is divisible by two and outputs either `True` or `False` based on the result as long as the input number is positive. Common misunderstandings were for instance that returning `True` means that control is returned to the main program, whereas returning `False` makes the subroutine start all over again. Some students thought that there is no output whenever the subroutine returns `False`.

The third category indicate that some students had difficulties deciding when a while loop stops, missing the last iteration. It should be noted that the loop will be executed once more after a negative value is input.

The fourth category is similar to the one for the first progress report. That is, students seem to be guessing, stating that the program outputs something totally different (in this case the input values) from what the subroutine returns.

The categories found were quite similar for both progress reports, and can be related to the SOLO taxonomy as presented by Lister et al. [9]. The first category (correct explanation) contains relational responses, whereas the second and third ones (indicating a misunderstanding) can be seen as containing explanations at the pre- or unistructural level. The fourth category (guessing) includes unistructural responses, which can also be seen as the ”speculative guesses” mentioned by Pennington [10].

4.2 Understanding of Individual Statements

In order to further analyze students' skills to read and understand code, we analyzed how they explained individual statements related to a set of programming topics. The explanations found were of the following types:

- *Correct* - the student explained the statement ”by the book”
- *Missing* - the student did not write any explanation for that given statement
- *Incomplete* - the student's explanation was correct to some extent, but still lacking some parts
- *Erroneous* - the student gave an explanation that was ”not even close” (for instance indicating a misconception)

Although the number of students giving correct overall explanations for the programs decreased from the first to the second report (as shown in section 4.1), the results presented in the diagram in Figure 1 indicate that at the same time the students became better at understanding individual statements: the number of correct explanations for individual statements increased while the number of incomplete or missing explanations decreased. This can, however, be seen as quite natural as one could - and should - expect students to gain a better understanding for individual topics as the course goes on and they become more experienced and familiar with the topics. We found no erroneous comments related to individual topics included in both reports.

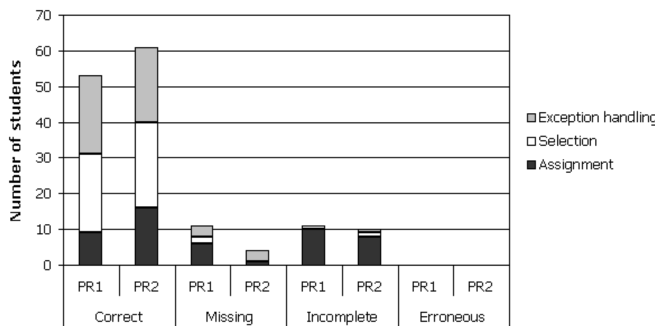

Figure 1: *The frequency of different types of explanations given by students for individual statements in the two progress reports.*

The second progress report introduced some new topics not present in the first one. The distribution of explanation types for these is illustrated in Figure 2. The data in the diagram reflect the previously mentioned difficulties related to subroutines returning Boolean values: almost half of the students gave incorrect explanations on this point. Interestingly, the other half of the students explained the returns correctly. This might imply that this topic (subroutine returns, at least for Boolean values) is something students either "get or do not get".

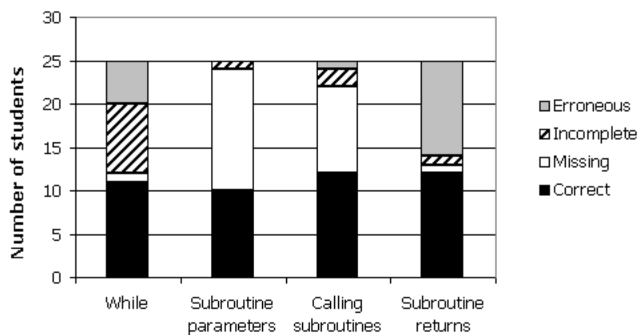

Figure 2: *The frequency of different types of explanations given by students for new topics in the second progress report.*

Many students did not explain subroutine calls or parameters, which makes it difficult to say anything about the perceived difficulty level of those topics. If all missing explanations indicate "erroneous explanations", the number of students not understanding subroutine calls and parameters is alarmingly high. On the other hand, if the missing explanations were due to students finding those aspects "self evident", and therefore not needing any explanation, the number of correct explanations for those topics would be high. When taking into account that the overall explanations to the code in the progress reports did not indicate any specific difficulties in calling subroutines with parameters, the latter explanation might be a bit closer at hand. These are, however, only speculations, and in our opinion the question of which aspects of subroutines make them difficult merits further investigation.

Having analyzed the explanations for both individual statements and entire programs, we can conclude that more students were able to correctly explain the program line by line than as a whole. This was found for both progress reports. When related to the levels in the SOLO taxonomy, most students were able to give correct explanations in multistructural terms, but only part of them did so relationally.

4.3 Difficulty of Topics

Apparently, assignment statements constituted the most common difficulty for students in the first progress report. However, this was not reflected in the students' own opinions on what they found difficult in the course at that time. Instead, they mentioned topics such as loops (n=4), the selection statement (n=2) and lists (n=3). Moreover, 40% of the students only gave an incomplete explanation for what `a = b` means, not mentioning values or variables, but stating for instance that "a becomes b" or "a is b". Clearly, such a student has some idea of what happens, but without mentioning values, this explanation is not exact enough.

In the second progress report, the problems students faced in the "trace and explain" questions (returns and subroutines) were in line with the difficulties they reflected upon in the other questions: almost half of the students stated that subroutines were most difficult. Some students still reported having problems with lists (n=2) and loops (n=2).

In the post course survey, students were asked to rate each course topic on the scale 1-5 (1 = very easy, 5 = very difficult). The results showed that the perceived difficulty levels were quite consistent with the corresponding results presented in our previous study [5]. Subroutines, modules, files and documentation were still regarded as most problematic (average difficulty of 2.8 - 3.2). There was, however, one exception: in the previous study, exception handling was also experienced as one of the most difficult topics (average of 2.9), but in the current study this was no longer the case (average of 2.1). The progress reports supported this finding: exception handling was not mentioned as a difficult topic at all, and nearly all students gave a perfect explanation for statements dealing with exception handling in the "trace and explain" questions. We were pleased to see this result, as we had made changes to the syllabus in order to facilitate students' learning of this particular topic. Exception handling was now one of the topics introduced at the very beginning of the course, together with variables, output and user input, and students got used to check for and deal with errors from the start. It thus seems as if the order in which topics are introduced does have an impact on the perceived difficulty level, as suggested by Petre et al. [12], who have found indications of topics being introduced early in a course to be perceived as "easy" by students, whereas later topics usually are considered more difficult.

5. CONCLUSION

In this paper we have introduced progress reports, and described how we have used these to analyze student understanding and progress during an introductory programming at high school level. Our initial experiences from using the reports are positive, as we feel that they provide important information during the course, which most likely would remain uncovered otherwise. The re-

ports can be used in various ways, and can be seen as a rather small active effort, with which one can collect valuable information that, if used wisely, can make a large difference for students learning to program.

Asking the students to fill out reports repeatedly throughout a course could, for instance, not only serve as a tool for continuous checkups of student progress throughout a course, but also as a starting point for individual discussions, in which the tutor/teacher and the student could go through the explanations and any potential errors. Naturally, such discussions would require extra resources in the form of teacher/tutor effort and time, which might not be available. A less demanding alternative would be for the teacher/tutor to only have short discussions with students that are evidently in need of help based on the report. At the same time they could try to find out where the difficulties truly lie - whether it is in the topics the student has written down, or somewhere completely different. Based on our findings, the latter would be most common, as the errors that actually occurred in students' code explanations did not always match the topics that were most problematic according to the students themselves. The difference was particularly evident in the first progress report, where most errors were related to assignment statements, but none of the students mentioned these as being difficult.

Some students seemed to be guessing when answering the "trace and explain" questions. In the future, we will add another question to the progress reports that ask the students to evaluate (e.g. on a given scale) how confident they are about their explanations. This will make it easier to distinguish between students truly believing in their answers and those merely guessing.

The results from the SOLO study presented by Lister et al. [14] are interesting, as they divide student responses into different SOLO categories. However, asking the students for both a multistructural and a relational response makes the data even more interesting, since it gives us two different responses for each program. These can be used to analyze how well the responses match for an individual student. As seen in the previous section, students were in general able to give perfect descriptions of the programs line by line, but only a fraction of these gave a perfect explanation of what the program did as a whole. This finding suggests that novice programmers tend to understand concepts in isolation, and is thus consistent with the results presented by Lister et al. [14] and with Pennington's idea of program vs. domain models [11].

As educators, we expect students to go through and learn from examples when we introduce a new topic. Doing so, the student's attention is on the construct, not on understanding how the given piece of code solves a particular problem. This means that we mainly support the development of a program model of understanding. For students to develop a more complete understanding of a program, we should also give them tasks and examples that facilitate them in the process of developing a solid domain model. The progress reports can be used as a feedback tool to help us evaluate how we are doing on this point.

The good results from introducing exception handling, a topic which was previously perceived as difficult, earlier in the syllabus were encouraging, and indicated that the order in which topics are introduced can make a difference. Since subroutines continue to be a problematic topic in introductory programming, we suggest that one would try to teach modular thinking and writing own, simple subroutines as one of the first topics in introductory programming courses.

6. ACKNOWLEDGEMENTS

Special thanks to Mia Peltomäki and Ville Lukka for collecting the data.

7. REFERENCES

[1] J. Biggs and K. Collis. *Evaluating the Quality of Learning - the SOLO Taxonomy*. New York: Academic Press, 1982.

[2] C. Corritore and S. Wiedenbeck. What Do Novices Learn During Program Comprehension. *International Journal of Human-Computer Interaction*, 3(2):199–208, 1991.

[3] L. E. Deimel and J. F. Naveda. Reading Computer Programs: Instructor's Guide and Exercises, 1990. Education materials. Available online: http://www.literateprogramming.com/em3.pdf. Retrieved August 29, 2006.

[4] V. Fix, S. Wiedenbeck, and J. Scholtz. Mental representations of programs by novices and experts. In *INTERCHI '93: Proceedings of the INTERCHI '93 conference on Human factors in computing systems*, pages 74–79, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press.

[5] L. Grandell, M. Peltomaki, R.-J. Back, and T. Salakoski. Why Complicate Things? Introducing Programming in High School Using Python. In D. Tolhurst and S. Mann, editors, *Eighth Australasian Computing Education Conference (ACE2006)*, CRPIT, Hobart, Australia.

[6] A. Haataja, J. Suhonen, E. Sutinen, and S. Torvinen. High School Students Learning Computer Science over the Web. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2001. Available online: http://imej.wfu.edu/articles/2001/2/04/index.asp. Retrieved August 29, 2006.

[7] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A Study of the Diffculties of Novice Programmers. In *ITICSE '05: Proceedings of the 10th annual ITiCSE conference*, pages 14–18, Capacrica, Portugal, 2005. ACM Press.

[8] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, 2004.

[9] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, 38(3):118–122, 2006.

[10] N. Pennington. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, pages 100–113, 1987.

[11] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

[12] M. Petre, S. Fincher, J. Tenenberg, et al. "My Criterion is: Is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. Technical Report 6-03, Computing Laboratory, University of Kent, UK, June 2003.

[13] D. Spinellis. Reading, Writing, and Code. *ACM Queue*, 1(7):84–89, October 2003.

[14] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. A. Kumar, and C. Prasard. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. In D. Tolhurst and S. Mann, editors, *Eighth Australasian Computing Education Conference (ACE2006)*.

[15] L. E. Winslow. Programming pedagogy, a psychological overview. *SIGCSE Bull.*, 28(3):17–22, 1996.