Student mistakes in an introductory programming course

Amitrajit Sarkar, Michael Lance, Mike Lopez, Robert Oliver, Luofeng Xu

Christchurch Polytechnic Institute of Technology
130 Madras Street,
Christchurch, New Zealand
+64 3 940 8000

sarkara@cpit.ac.nz, lancem@cpit.ac.nz, lopezm@cpit.ac.nz, oliverr@cpit.ac.nz, xul @cpit.ac.nz

ABSTRACT

Learning to program is a challenging task for novice learners. This study aimed to investigate students' concepts as they were being formed. To capture these, we chose to focus on students who made some mistakes in basic concepts. Our study sought to capture students' conceptions at a very early stage in their study: five weeks into an introductory programming course. We invited students who did not pass an early mastery test at their first attempt to participate in a diagnostic and remedial session. In this session, the teaching team carried out one-on-one interviews with students to diagnose any misconceptions the students exhibited and devise individual remedial learning. The teaching team documented these interviews and these formed the basis of our phenomenographic analysis. Our main finding was that the lack of success in the test was attributable more to application of process than to conceptual misunderstandings. We also found that the technique of inviting students who do not succeed in a test to participate in a in-depth diagnostic interview and one-on-one remedial instruction was useful, even though no major misconceptions or alternative conceptions were identified.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education]**: Computer Science Education – *programming misconceptions and mistakes.*

General Terms

Human Factors.

Keywords

Misconceptions, alternative conceptions, novice programming, programming mistakes.

1. INTRODUCTION

Learning to program is a challenging task for novice learners (Eckerdal, Laakso, Lopez, & Sarkar, 2011). Beginning programmers need to master a wide range of concepts (du Boulay, 1989) and consequently, it is not surprising that many wide scale studies have found that novices are struggling in the early stages of this learning process. Previous research has studied students' learning of skills (Lister, et al., 2004; McCracken, et al., 2001; Tenenberg, et al., 2005) as well as concepts (Fleury, 2000; Boustedt, 2010).

The present research takes its starting point from a phenomenographic study on novice students' understanding of the concepts object and class (Eckerdal, Laakso, Lopez, & Sarkar, 2011). We believe that studying students as concepts are being formed can give us insights into the challenges they face as they struggle to master the necessary concepts. One way of identifying potential students who are in the process of forming concepts is to look at students who make mistakes in an assessment. Thus, we can use their mistakes in a sense as a lens, to capture their ideas as they are forming. Our study sought to capture students' conceptions at a very early stage in their study: five weeks into an introductory programming course.

We believe this is important because a beginning programmer has to master many concepts and these build on each other progressively as the course unfolds. Unless there is a solid foundation in the early concepts, a student may struggle throughout the course (Robins, 2010).

The rest of this paper is organised as follows. In section two, we review related work. In section three, we describe our method. In section four, we present our findings. In section five, we discuss our findings. Finally, we present our conclusions in section six.

2. RELATED WORK

Students' misconceptions are frequently reported in studies on students learning to program. Most introductory programming educators will recall when they were completely perplexed by how the novice programming students expressed their understanding of a critical programming concept. Ragonis and Ben-Ari present a large study with high school students learning to program. Their study includes detailed lists of difficulties and misconceptions related to several concepts in object-oriented programming (Ragonis & Ben-Ari, 2005). Holland, Griffiths and Woodman claim that misconceptions of basic object concepts can be hard to shift later and that such misconceptions can act as barriers through which later all teaching on the subject may be inadvertently filtered and distorted (Holland, Griffiths, & Woodman, 1997).

Sanders and Thomas describe a close examination of student programs from an introductory programming course, in which they found evidence of misconceptions. Among other things they found difficulties in distinguishing between classes and objects, and in modelling (Sanders & Thomas, 2007).

Spohrer and Soloway studied novice Pascal students and investigated whether or not most novice programming bugs arise because students have misconceptions about the semantics of particular language constructs (Spohrer & Soloway, 1986). The authors found that for most of the bugs investigated that was not the case. Bayman and Mayer report on a study on beginning BASIC programmers' misconceptions of statements they had learned (Bayman & Mayer, 1983), and Fung, Brayshaw and du

This quality assured paper appeared at the 4th annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2013) incorporating the *26th Annual Conference of the National Advisory Committee on Computing Qualifications*, Hamilton, New Zealand, October 6-9, 2013. Mike Lopez and Michael Verhaart, (Eds).

Boulay report on novices' misconceptions about the interpreter in Prolog (Fung, Brayshaw, & du Boulay, 1990) .

These papers provide several viewpoints on the characteristics and common misconceptions of novice programmers that should be considered when designing approaches for programming education. These sources conclude, for example, that novice programmers are typically limited to surface knowledge of programs. They often approach programming line-by-line rather than using meaningful program structures. The knowledge of novices tends to be context specific and they also often fail to apply the knowledge they have obtained adequately. They may know the syntax and semantics of individual statements, but do not know how to combine them into valid programs.

2.1 ALTERNATIVE CONCEPTIONS

We use the term alternative conception to highlight conceptions that do not align with computer science, but are nevertheless rational and typically based on prior ideas that are useful in non-computer science contexts. As Smith and colleagues comment, "We suggest that misconceptions, especially those that are most robust, have their roots in productive and effective knowledge" [16, p.124]. Because such alternative conceptions are useful in many contexts, we would expect them to be difficult to displace. Indeed, it is questionable whether such an attempt should be made. As Smith and colleagues further note:

> Instruction designed to confront students' misconceptions head-on (e.g., Champagne et al., 1985) is not the most promising pedagogy. It denies the validity of students' conceptions in all contexts; it presumes that replacement is an adequate model of learning; and it seems destined to undercut students' confidence in their own sense-making abilities. Rather than engaging students in a process of examining and refining their conceptions, confrontation will be more likely to drive them underground. [16, p.153-4]

Moreover, a commitment to allowing the student enough freedom and time to explore an idea is essential. As Duckworth noted,

> Teachers are often, and understandably, impatient for their students to develop clear and adequate ideas. But putting ideas in relation to each other is not a simple job. It is confusing; and that confusion does take time. All of us need time for our confusion if we are to build the breadth and depth that give significance to our knowledge. [4, p.82]

From these two perspectives, we should not attempt to expose and confront misconceptions too readily. Nevertheless, we believe that it is helpful to us as educators if we can develop a better sense of the alternative conceptions held by students.

3. METHOD

The context of our study is a compulsory introductory software engineering course, at level 5 in the New Zealand Qualifications Framework (NZQA, 2011), which is taught over one semester as part of a three year computing degree. In the first five weeks of the course, the Scratch programming language is used in tutorial sessions to introduce fundamental concepts that underpin programming: variables, sequence, selection, repetition, recursion, and message passing. At the end of this segment, students take a test which assesses their ability to trace and analyse code. The test has six questions; for each question, the student has to predict correctly the output of a short program. Students are expected to carry out hand execution of the code to determine this output. However, a student who can read and understand the program as a whole could bypass the desk-checking process and determine the output directly from their understanding. This test has a mandatory pass requirement. Students who fail the test are allowed one resit, but those who fail the resit cannot pass the course overall.

In this study, we focused on those students who did not pass the test. We were interested in exploring in more detail the reasons for failure. Our interest was motivated from two points of view. First, as part of our normal teaching practice, we were interested in what interventions we could take to enhance the chance of students passing the test on a resit. Second, from a research perspective, we were interested in whether we could derive any general lessons or principles from what we found. In this regard, the tacit theory underpinning our teaching is the use of a constructivist learning environment with heavy scaffolding (Wood, Bruner, & Ross, 1976). In the absence of an explicit theory of programming mistakes, we used an exploratory approach based on interpretative phenomenological analysis (Smith, Flowers, & Larkin, Interpretative Phenomenological Analysis: Theory Method and Research, 2009) of a discourse between student and teacher.

We approached students who did not pass the test at their first attempt and invited them to participate in a diagnostic and remedial session. All students chose to participate in this session. Members of the teaching team were assigned to students and worked with them on a one-to-one basis. The assigned teaching team member reviewed the student's test answers with the student and asked the student to demonstrate how they carried out the desk-checking tasks that gave rise to the answer they had submitted. Each lecturer produced a written report summarising each of these sessions with students and these reports were shared and discussed among the teaching team and used as the source for the analysis in this research.

4. FINDINGS

In this section, we present the lecturer summaries of three of these student interview sessions. We believe these are representative of the overall range of findings.

Student A was a mature female student. She struggled to relate to the course and after failing the resit, withdrew from the course. We present our findings for this student in Section 4.1.

Student B was a mature female student, who also struggled at first to relate to the course. However, she reacted well to the remedial instruction and subsequently succeeded in the test resit and the course overall. The investigation of this student is presented in section 4.2.

Student C had no difficulty understanding the material and related well to programming. However, he felt that using formal desk-checking was too time consuming and looked for shortcuts. Nevertheless, he reacted well to remedial instruction; he subsequently achieved 100% on the test resit and a high grade overall for the course. Section 4.3 presents the investigation of this student.

4.1 Student A

The student was given six different segments of program code to study (under examination conditions) and determine the correct outputs. In the first attempt at answering the questions 5 out of the 6 were answered incorrectly. If one considers success in determining the correct answer is to apply the principles of desk checking, then those principles that were demonstrated and applied during class lectures did not allow the student to achieve this result. The question is why?

The student was asked to study the questions and explain her understanding of what the code was designed to achieve to a staff member. In all cases the student version of the structured desk check taught in class was used. That is to say the methodology taught was not used but a student interpretation of it was. The same general approach and methodology was used with each of the six problems involved.

The procedures used by the student involved an undisciplined method of recording the outcomes of each segment of program code. In fact the procedures were so undisciplined that the student wrote specific outcomes and results obtained from program code execution in random locations on paper and became totally confused as to where the latest and last result was recorded. That is to say the inability (or unwillingness or failure to identify and replicate, use and apply the structured methodology taught in lectures appeared to hinder the ability to derive the correct solution to the problem.

As an observation, when the student commenced using the standard desk-checking procedures an improvement in the ability to correctly identify code segment outcomes was obtained at a faster and more accurate rate.

4.2 Student B

The academic history of this student demonstrated consistently high academic achievement. However, surprisingly, the student did not pass the test of applying desk-checking with the required minimum mark. To find out the reason why the student failed, the following investigation was implemented.

First, the student was asked to identify which questions the student did not answer correctly in the test. The student did not know. Then, a staff member indicated the incorrect questions and asked the student to re-calculate these ones. The staff member observed how the student applied desk-checking.

It was found that the student did not use the version of desk-checking taught in class. Instead, the student used a different one. The student's own version of desk-checking clearly involved a less disciplined method of recording the outcomes of each segment of program code. That created an unclear trace of calculation. It confused the student when the student tried to retrieve the temporary values of variables during the calculation. It also prevented the student to implement an effective double check after completing the calculation. Moreover, it was quite difficult for the student to identify which step of the calculation was wrong, even if the student had doubt about calculated values and/or results.

Eventually, the staff member demonstrated how to use the desk-checking taught in class to solve one of the questions which the student did incorrectly in the test. The student was then asked to recalculate all incorrect questions by applying the class version of desk-checking. All questions were solved correctly at a faster and more accurate rate.

Later, the student passed the re-sit test of desk-checking by correctly answering 5 out of 6 questions.

4.3 Student C

This student had a good grasp of programming concepts and subsequently passed the introductory programming paper with a high grade. However, surprisingly, the student did not initially pass the test with the required minimum mark. To find out the reason why the student failed, we carried out the following investigation.

We asked the student to show us his desk-checking method for the questions that he answered incorrectly in the test. Interestingly, he was able to demonstrate a flawless performance when desk-checking each of these questions. When we probed further, he mentioned that he has done desk-checking for the first two questions but, because he found the desk-checking process tedious, he decided to guess the answers to the remaining questions rather than meticulously doing further desk checking. The consequence of this guessing was that he ended up scoring lower marks than he would have achieved if he had applied his desk-checking technique.

In summary, this student had a perfect understanding of desk-checking and a sound grasp of logic. He achieved 100% in his test resit.

5. DISCUSSION

We begin by analysing our intervention from the perspective of attribution theory (Wiener, 1992). Weiner's theory analyses attribution across three dimensions: locus of control (internal and external), stability, and controllability. How a student attributes the reason for lack of success is important because it affects the student's motivation to remedy the lack of success by engaging in further learning. For example, an attribution to luck which is external and uncontrollable would create little motivation to learn. Similarly, an attribution to internal stable factors such as personality would create little motivation. However, all of the students attributed the reason for their lack of success to an internal locus of control, believed that the outcome could be modified, and believed that this was under their control. This combination suggests that the application of remedial techniques could be effective and supports the approach of offering students who did not succeed on their first attempt the option of attending a diagnostic session with subsequent remedial instruction.

A student is also affected by their motivation for the subject. Subsequent to the experience presented in this paper, the students took different study paths. Student A has a previous degree outside the computing field and was doing the course to develop her understanding of computing rather than to gain accreditation. She has not yet decided on further study. Student B subsequently chose a study path leading to a major in Information Systems. Student C chose a study path leading to a major in Software Engineering. Race (Race, 2010) uses the terms *wanting* for intrinsic motivation and *needing* for extrinsic motivation. At a superficial level, we could say that student C wanted to learn programming and student B needed to pass this compulsory course, but student A neither wanted nor needed nor needed to learn programming. However, this probably underestimates the complexity of human motivation.

We can add two more perspectives. First, desk-checking and tracing is included in the course primarily as scaffolding to support understanding (Wood, Bruner, & Ross, 1976). As a student's understanding of programming grows, there is less need for such scaffolding and it may be withdrawn. However, the need for scaffolding will vary from student to student and thus the timing of withdrawing scaffolding needs to be considered carefully. Analysis of student C's performance suggests that the scaffolding may have been withdrawn prematurely in his case.

A second perspective is that of cognitive load theory (Sweller, Cognitive load during problem solving: Effects on learning, 1988). Although scaffolding techniques, such as desk-checking or hand execution are introduced as an aid to student understanding, there is a risk that the use of formal techniques might become just

another thing for a student to learn and master. Effective instructional design requires careful consideration of the cognitive load imposed by a technique when considering the benefits that arise from its use (Sweller, Cognitive architecture and instructional design, 1998).

Our experience with this exercise has led us to believe that the technique of carrying out in-depth diagnostic interviews with students who do not succeed in a test, and then engaging in one-on-one remedial instruction can be beneficial. In short, such an early intervention works, but it may not work for everyone. A successful intervention is likely to require both that the student attributes the cause of lack of success to internal, malleable, controllable factors and that they are motivated to succeed in the course, whether this motivation is intrinsic or extrinsic.

We also believe that the nature of feedback given to a student is critical. Such feedback needs to be personalised and given in a form that can be readily acted upon by the student.

6. CONCLUSION

We found that the technique of inviting students who do not succeed in a test to participate in an in-depth diagnostic interview and one-on-one remedial instruction was useful, even though no major misconceptions were identified. Indeed, we found that the lack of success in the test was attributable more to application of process than to conceptual misunderstandings or alternative conceptions. However, our diagnostic interviews focused closely on performance in the test and it is possible that a more broadly focused interview would have discovered alternative conceptions.

Our study was carried out with a single cohort of students in a single course. Accordingly, we do not believe that it is appropriate to generalise from these findings to other contexts. Nevertheless, we believe that the approach to intervention could be tried and evaluated in other contexts.

We plan to continue to explore and refine the use of this technique. In particular, we will use a broader approach in our diagnostic interviews to attempt to elicit alternative conceptions.

7. REFERENCES

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM, 26*(9), 677–679.

Boustedt, J. (2010). On the road to a software profession: Students' experiences of concepts and thresholds [PhD thesis]. Uppsala University.

du Boulay, D. (1989). Some difficulties of learning to program. In *Studying the Novice Programmer* (pp. 283–299). NJ: Lawrence Erlbaum.

Duckworth, E. (1987). *The "having of wonderful ideas" and other essays on teaching and learning.* New York: Columbia University, Teachers College.

Eckerdal, A., Laakso, M., Lopez, M., & Sarkar, A. (2011). Relationship between text and action conceptions of programming: a phenomenographic and quantitative perspective. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education (ITiCSE '11)* (pp. 33-37). New York: ACM.

Fleury, A. (2000). Programming in Java: student constructed rules. *SIGCSE Bulletin, 32*(1), 197-201.

Fung, P., Brayshaw, M., & du Boulay, B. (1990). Towards a taxonomy of novices' misconceptions about the prolog interpreter. *Instructional Science, 19*(4-5), 311-336.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education.* San Jose, CA.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multinational study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David, et al. (2001). A multinational, multiinstitutional study of assessment of programming skills of first-year cs students. *SIGCSE Bulletin, 33*(4), 125–180.

NZQA. (2011). The New Zealand Qualifications Framework [Version 2]. Wellington, NZ: NZQA.

Race, P. (2010). *Making Learning Happen.* London: SAGE Publications.

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts. *Computer Science Education, 15*(3), 203–221.

Robins, A. (2010). Learning edge momentum:A new account of outcomes in CS1. *Computer Science Education, 20*(1), 37–71.

Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented cs1 programs: concepts and misconceptions. *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education.* New York, NY.

Smith, J., Disessa, A., & Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences, 3*(2), 115-163.

Smith, J., Flowers, P., & Larkin, M. (2009). *Interpretative Phenomenological Analysis: Theory Method and Research.* London: Sage.

Spohrer, J. C., & Soloway, E. (1986). Alternatives to construct-based programming misconceptions. *SIGCHI Bulletin, 17*(4), 183–191.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257–285.

Sweller, J. (1998). Cognitive architecture and instructional design. *Educational Psychology Review , 10*(3), 251–296.

Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T., Chinn, D., et al. (2005). Students designing software: a multi-national, multi-institutional study. *Informatics in Education, 4*(1), 143-162.

Wiener, B. (1992). *Human Motivation: Metaphors, Theories and Research.* Newbury Park, CA: Sage Publications.

Wood, D. J., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychiatry and Psychology, 17*(2), 89-100.