

USING VIDEO TO EXPLORE PROGRAMMING THINKING AMONG UNDERGRADUATE STUDENTS

Carol Wellington and Rebecca Ward
Shippensburg University
Shippensburg, PA

ABSTRACT

Phenomenography is a qualitative method that primarily has employed interviews of students to explore variations in how they experience some phenomenon. This study is an experiment in phenomenography in which the basic data (the interview) is replaced by video of students working on a computer science project with the goal of exploring variations in how students approach programming activities and laying the groundwork for more extensive study. While this is a preliminary study, early analysis shows that the addition of videos enriches the conclusions that can be made and further exploration of this technique is warranted.

1 INTRODUCTION

Phenomenography is an empirical, qualitative research method designed to determine differences in how people experience specific phenomena (Marton & Booth, 1997). In other words, phenomenography is a way of looking for qualitatively different conceptualizations of an experience in a group of people. The basic structure of a phenomenographical study is formal interviews of subjects that are transcribed. Those interviews are then analyzed to look for categories in which the participants vary and exploring those variations. In computer science, this method has been used to explore student understanding of programming thinking (Eckerdal & Berglund, 2005) and how students learn to program (Govender & Grayson, 2008). Lister (2003) makes a strong argument for using phenomenography as a way to explore to what extent students understand computer science topics. In the current study, we use phenomenography as a base from which to explore programming thinking as a developmental construct. We are particularly interested in the qualitative changes that occur in programming thinking as computer science students progress through their undergraduate programs.

A key component of phenomenography research is that data is collected from the perspective of the student rather than the researcher and involves determining varying ways that students interact with phenomena. While interviews are often used to collect this data after the fact, we employed audio and video technology in order to capture “real time” thinking and programming. Students were asked to “think out loud” during a

programming assignment while their voices and computer screens were recorded. Our sample size was intentionally small in order to determine the validity of the research method and to begin a qualitative exploration of themes to be used in coding for future research.

2 EXPERIMENT DESIGN

We are interested in how video phenomenography might be used to learn about how students' problem solving skills mature throughout their undergraduate education. A better understanding of the way that problem solving abilities mature could lead to the development of curricular materials that assist that maturational process.

For this preliminary study, we wanted to study a broad cross-section of computer science majors, so we put out a call to all of our majors for volunteers. In that call, we explained the purpose of the study and asked students for one hour in which their efforts on a programming activity would be recorded. As a result of that call, we recorded students with these characteristics:

Student	Description	# of CS courses completed
Two Freshmen	A pair of female freshman working together	1
Freshman 1	A male freshman with significant pre-college experience	1
Freshman 2	A female freshman	1
Sophomore 1	A male sophomore	2
Junior 1	A male junior	9
Senior 1	A male senior ready to graduate	11

The laboratory activity for this study was a Computer Science I activity that is usually assigned about $\frac{3}{4}$ of the way through the semester. The goal of the activity is translating zip codes into bar codes and it requires conversion of ints to Strings, breaking an int into its component digits, and use of arrays, loops, and conditionals. The instructions for the lab include information about bar codes and specific instructions that lead the students through test-driven development of the required class. For example, the lab is specific about which border cases need to be tested, but it does not give instruction on how to write the code to make those tests pass. It is important to note that all of our students have been exposed to unit testing with JUnit, but only our freshmen and sophomores have been explicitly instructed in test-driven development.

The lab was given to the students before their recording time and they were instructed to read it before they were recorded (not all of the students did that). At their recording time, the students used a Mac and Eclipse to work on the lab and were instructed to describe what they were doing and why they were doing it. Snapz recorded the contents of their screen and their voices as video.

3 VIDEO ANALYSIS STRATEGY

Each of the videos was transcribed into a spreadsheet with 15 seconds for each row. In addition to what the student said, whether the student was doing each of these activities was gathered for each interval:

- reading?
- re-reading?
- following TDD?
- writing tests?
- writing code?
- running tests?
- adding debug printouts?
- using the debugger?
- tests were red?

A computer science faculty member also watched the videos and added comments about what the student appeared to be doing, any misconceptions the student seemed to have, and any other relevant comments.

4 PRELIMINARY RESULTS

Our preliminary analysis has shown a number of areas in which the strategies used by students differ with their experience.

4.1 Irrelevant changes

The lower division students made some changes to their solutions that seem to imply some basic misunderstandings of how some language constructs work.

Both “2 Freshmen” and “Sophomore 1” added extraneous else clauses when their tests failed. In both cases, they made changes like this:

Original Code Structure	Modified Code Structure
<pre>if (condition) return val1; return val21</pre>	<pre>if (condition) return val1; else return val2</pre>

The addition of the “else” has no effect on the behavior of the code, but, in both cases, the students re-ran their tests hoping they had fixed the problem. This appears to show a lack of understanding of how “return” affects the flow of execution in the method.

“2 Freshmen” also tried re-arranging a sequence of nested conditionals:

Original Code	Modification
<pre>if (zipCode < 100000) return "0" + zipCode; else if (zipCode < 10000) return "00" + zipCode; else if (zipCode < 1000) return "000" + zipCode; else if (zipCode < 100) return "0000" + zipCode; else if (zipCode < 10) return "00000" + zipCode;</pre>	<pre>if (zipCode < 10) return "00000"; else if (zipCode < 100) return "0000" + zipCode; else if (zipCode < 1000) return "000" + zipCode; else if (zipCode < 10000) return "00" + zipCode; else if (zipCode < 10000) return "0" + zipCode;</pre>

Since these conditions were mutually exclusive, their order had no effect on the external behavior of the method. The students made this change and, even when it did not change the results of the tests, they did not comment on the fact that their effort made no difference. In fact, it took another three and a half minutes for them to see the off-by-one errors in their strings. Upper division students did not make these types of mistakes.

4.2 Search for Help

When students were unsure about how to proceed, they looked for help in a variety of places:

“2 Freshmen” and “Freshman 2” looked back at the lab for clues. Since the lab contained little information on syntax (except for an explanation of the modulus operator), this probably means that their troubles lay in formulating a solution more than in finding the specific syntax details. In fact, when “2 Freshmen” were trying to break the zip code into digits, they spent twelve minutes writing a variety of attempts at the solution using the syntax correctly, but were unable to put the pieces together into a solution that worked. In that 12 minutes, they went back to the lab for clues five times. In the middle of this time, one of these students gave another strategy for getting help when she said, “This is when I go to somebody else who knows more.”

All of the students, to one degree or another, used Eclipse’s ability to recall methods when they were stuck. For example, “Junior 1” and “Senior 1” both had difficulty remembering how to convert an int to a String. Initially, they both typed the name of the variable followed by “.” and waited for clues from Eclipse. Only after Eclipse gave no possibilities did they remember that ints are primitives, so there isn’t an existing method to make the conversion.

Both “Junior 1” and “Senior 1” used Google to find strategies. In fact, “Senior 1” was very quick to go to Google; when Eclipse gave no clues, he immediately brought up a browser and did a very quick search to find a solution. He went so far as to say, “Where’s my Google machine? That’s the best problem-solving skill. If I had to say one thing saves me every time . . .”

4.3 Off-By-One Errors

There is some evidence that teaching students to be aware of specific types of errors may help them prevent and/or repair them. In this particular lab, off-by-one errors were a common point of discussion.

“2 Freshmen” expressed an awareness of off-by-one errors by mentioning them when they were writing a loop, but they didn’t draw any conclusions about whether or not they currently had such an error. Later, they made an off-by-one error where a loop goes one more time than necessary, but it had no effect on the external behavior of their method and they did not catch it.

“Sophomore 1” was clearly aware of off-by-one errors and tried almost random things to fix them. The lab required the addition of leading zeros when outputting the zip code when it is stored as an integer. His initial attempt was to check if the zip code was less than five, confusing the length of the integer (in digits) with its value. When that didn’t work, he ran through this sequence of conditions: < 6 (runs the tests), ≤ 4 (runs the test), ≤ 5 , < 5 (where he runs the tests commenting that he knows that’s the first condition he

tried), (at this point he makes it print out the zip code which is 1234) ≤ 4 . He was so focused on the off-by-one error that it took him almost four minutes to recognize that he was looking at the integer's value instead of the number of digits it contained. He made a similar sequence of changes to a loop variable later in the lab. In both cases, the error he was making was not an off-by-one error.

“Junior 1” also wrestled with off-by-one errors. At one point in breaking the zip code into digits, he got an array out of bounds exception. He talked about the potential of having an off-by-one error: “It's not just an off by one error because at the end this will still go up to four and I'll still have an out of bounds exception.” He was aware of the possibility of an off-by-one error, but could reason about why that isn't his problem without writing the code.

4.4 Code Scribbling

The most interesting thing we've seen in these videos is what we call “code scribbling.” “Sophomore 1” is the best example of this technique. As he was trying to code his solution, he typed out a series of attempts. Often, he ran into a syntactical reason why his solution wouldn't work and then backed up and started again. However, sometimes he reached a solution that was syntactically correct, looked at what he had, said, “That won't work” and backspaced to delete that solution and try again. It appears that he has to see his incorrect code before he can predict that its behavior is incorrect.

Given our observations of “Sophomore 1,” we looked for similar behavior in the other students. “2 Freshmen” also progressed through a series of solutions, but they were not as adept at predicting that the code would fail. They had to see the tests fail before realizing that the code wouldn't work.

“Junior 1” also progressed through a series of solutions. However, he described each solution (without writing any code) and predicted that the solution would or would not work without needing to see the code written out.

These observations provide evidence of a progression in the students' ability to imagine a possible solution and evaluate its efficacy:

1. Initially, students have difficulty predicting how a piece of code will work and depend on running the code to see what it does.
2. Later, after they write a piece of code, they can predict that it won't work, but they have to finish writing it first. In other words, they can only imagine its behavior after they have written it.
3. Finally, students can imagine a solution and detect that its behavior will be incorrect without writing any of the code.

5 CONCLUSIONS/FUTURE WORK

This work is a preliminary study from which few hard conclusions can be drawn. The goal was to explore the use of applying phenomenography to videos as a tool for understanding how students' problem solving skills mature. We have seen variations in problem solving strategies that appear to show a cognitive difference between students with different levels of experience. In particular, their ability to imagine the behavior of code seems to improve with experience. However, more work is required to validate these conclusions.

The videos contain information that we have not yet analyzed. For example, we have data about the percentage of time that their tests were red, whether they were following test-driven development, and their use of the debugger and debug printouts. However, we have not fully analyzed that data. We will use that data and further analysis of the videos to explore these questions:

1. How close to the actual defect do they place the breakpoint/debug printout?
2. To what extent do they rely on the instructions in the lab as opposed to going ahead based on the description of the problem?
3. Do they recognize that a strategy for solving the problem is overly complex and back up to look for a new strategy?

The early results of this study validate the potential of video in a phenomenography based study of programming thinking and, therefore, warrant a more complete, well-structured experiment. In that experiment, we would like to make two improvements: enhance the video component of the study and modify the way the students are selected and studied.

In the current videos, there are points where it is difficult to ascertain what the students are doing because they are quiet and not typing. In future video phenomenography experiments, we would like to add a camera pointed toward the student to be able to better assess what they are doing. This would also capture non-verbal communication that may signal early progress or frustration.

The current experiment has one fundamental weakness: it is assuming that problem-solving skills have matured as students progressed through our curriculum. However, we have a significant retention problem; there is a chance that the students who persist into our upper division courses enter the program with better problem-solving skills. Therefore, we would like to run a more disciplined experiment focusing on how problem-solving skills mature in the first year of collegiate programming. In that experiment, a set of freshman computer science majors would be randomly selected. Each would be filmed for one hour each week through our Computer Science I and Computer Science II courses. The goal of that experiment would be to look for trends in how their problem-solving strategies evolve through that critical year.

REFERENCES

- Eckerdal, A., & Berglund, A. (2005). What does it take to learn 'programming thinking'? *ICER'05*, October 1-2, Seattle, Washington.
- Govender, I. , & Grayson, D.J. (2008). Pre-service and in-service teachers' experiences of learning to program in an object-oriented language. *Computers & Education*, 51(2), 874-885.
- Lister, Raymond. (2003). A Research Manifesto, and the Relevance of Phenomenography. *Inroads – The SIGSCE*, 35(2), 15 – 16.
- Marton, F., & Booth, S. (1997). *Learning and Awareness*. Lawrence Erlbaum Associates, Mahwah, NJ.