Novice Programmers' Reasoning about Reversing Conditional Statements

Cruz Izu
The University of Adelaide
Adelaide, Australia
cruz.izu@adelaide.edu.au

Claudio Mirolo
University of Udine
Udine, Italy
claudio.mirolo@uniud.it

Amali Weerasinghe
The University of Adelaide
Adelaide, Australia
amali.weerasinghe@adelaide.edu.au

ABSTRACT

Abstraction and the ability to understand and apply abstract concepts, such as program behaviour, causes novice programmers to struggle with both introductory and advanced programming courses. Thus, evaluating students' ability to reason about small programs should be an important topic for CS education. Recent work has explored the strategies used by a sample group of students to reason about reversibility. That study showed many students failed to correctly reason about reversing a seemingly simple conditional statement. This work extends that study by assigning that exercise to two cohorts of novice programmers as part of the final exam paper.

We measured and analysed their ability to reason and/or reverse conditional statements. Reasoning about reversibility requires students to have a mental model of the state, which in the case of conditional statements includes (1) identifying the changes of the assignment, (2) evaluating any adjustments to its condition and (3) recognizing potential overlaps over the two execution flows. The evaluation showed 28% of them had that correct model, while an additional 23% could write the code but failed to identify the overlap.

KEYWORDS

programming; reasoning; reversibility; state.

ACM Reference format:
Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. 2018. Novice Programmers' Reasoning about Reversing Conditional Statements. In *Proceedings of SIGCSE 2018, Baltimore, Maryland, USA – Published article:, February 2018 (SIGSE)*, 6 pages.
https://doi.org/10.1145/3159450.3159499

1 INTRODUCTION

The ability to reason about program behavior is considered to be an advanced ability that CS2 students should develop [12]. The coverage of program behaviour in CS1 is usually focused on practical abilities such as tracing and testing. Little guidance is given on how to reason on the program behaviour as a whole, i.e. by considering *any* possible state or dataflow caused by program execution.

Recent work has investigated on program comprehension of CS2/CS3 students by exploring an abstract concept related to coding: *reversibility* [5]. Being able to determine reversibility of a code segment as an indicator for mental development of novice programmers was first proposed by Lister [7]. Subsequently, Teague [16] evaluated the **coding** ability of a large (n=603) group of CS1/CS2 students to reverse a small program. Later on, Izu[5] asked students to identify reversibility in small fragments of code, thus testing students' ability to **reason** systematically about state transformation. However, to test their full understanding we should combine both **reasoning** and **coding** in one task.

Our study will use a task that requires students to reason and code about the reversibility of conditional statements. We choose that task for three reasons: (1) it relates to the core concept of state that they have been exposed to, (2) it will force them to reason about the overlaps of the two paths of a conditional statement and (3) a similar task appeared to be a stumbling block in [5].

The exercise was part of the final examination for two cohorts of first year students at different countries; thus the task had to be simple and clear. We covered both how students reason about it, and how well they can code it in the same task as follows: firstly we asked them to decide if a conditional statement was reversible or not; secondly they had to justify their decision by providing the code that will undo the state if their answer was *Yes*, or an example of a state (more specifically, the value of a variable) that cannot be reversed if their answer was *No*. Deciding on reversibility force dstudents to reason about program behaviour in terms of possible changes to the program state.

This study addresses the following research questions:

- Can novice programmers reverse a conditional statement?
- How comprehensively can novice programmers envisage the impact of conditionals in state variables?

The main contribution of this study is to evaluate for the first time the understanding of novice programmers of the overall effect that *the alternative paths* of a conditional statement can have on program state.

The rest of the paper is organised as follows: section 2 reviews related work, section 3 describes the methodology used, section 4 presents the results, section 5 discusses the findings, followed by conclusions.

2 RELATED WORK

Reversibility is the mental process of determining if a set of events can be undone to return to the original state. In Piaget's theory of cognitive development, reversibility is a key step toward more advanced thinking: it represents the change from using and manipulating symbols to extending their meaning and making an increased use of logical thinking. Moreover, according to a neo-Piagetian perspective, corresponding learning stages are relevant independently of the age when approaching new knowledge domains [15].

There is a limited number of studies that focus on understanding program behaviour as a whole. Pennington [11] investigated

Pre-print of the paper (accepted manuscript) for the institutional repository and not for redistribution. See terms of the ACM Copyright Transfer Agreement.
SIGSE, February 2018, Baltimore, Maryland, USA – Published article:
https://doi.org/10.1145/3159450.3159499

different types of information available when analyzing a program: Operations, Control flow, Dataflow, State, and Function. Note these concepts are inter-related, for example Dataflow is about transformations that data objects undergo during execution; these transformations result from the combination of control flow and operations that change the object state. From this perspective, our reversibility task relates to both Dataflow and State. The task asks students to consider how to undo a given dataflow transformation, so they return the variable to its original state.

Of course, there is a large number of CS education papers that discuss various aspects of program behaviour such as code comprehension [3], ability to read code [9] and mental models [2]. However, their emphasis is not on reasoning about program behaviour as a whole, but on supporting students to learn complex concepts and improve their programming skills. For example, Ragonis et al. [12] discussed the difficulties students have in comprehending program flow under the object oriented paradigm, and suggested support required, such as visualisation, to improve comprehension. Lister et al., focused on reading code segments and being able to express the program focus or goal in one sentence, i.e. the ability to do chunking [8].

In regards to studies that explore the behaviour of conditional statements, it is worth mentioning what Cherenkova et al. report [4] while discussing the most frequent sources of students' problems with conditionals:

> A significant fraction of the students do have trouble with basic structural issues of syntax and indentation. [...] However, a significant number [...] exhibit the common errors of failing to check the border condition [26.7%] or reversing the conditional [28.5%].

3 METHODOLOGY

3.1 Task

Valid assessment of higher order thinking skills requires that students be unfamiliar with the given task. Accordingly, on their tutorial or class exercises students were not exposed to reversibility. At different points of the course, however, they have been asked to evaluate the outcome of small pieces of code. Thus, they should have sufficient prior knowledge to complete the proposed task. For the purpose of our investigation, we selected a mixed method approach that combines the three item/task formats useful in measuring higher order skills [6]: firstly, students use *selection* to identify whether a code segment is reversible or not, then they either use *explanation* by providing an example of a state not being reversible, or they use *creation* by writing the code to undo the reversible command. The exam question is shown in Figure 1. Note in the exam paper the question is preceded by a basic introduction on reversibility with examples, not shown here due to space limitations.[1]

The code block inside each conditional statement is simple and clearly reversible. Adding or subtracting a constant value is easily reversed, as explained in the introduction to the task. The key insight required to analyse the reversibility of the given statements is that there are two flows of control whose outcomes may overlap:

[1]Both versions will be included as an additional file to the final paper.

Analyze the following code fragments and determine if they are reversible:

- if your answer is *Yes, reversible* (i.e. the command is reversible), write a piece of code which restores the original state of the program variables.
- If your answer is *No* (i.e. it is not always possible to undo the effect), provide an example where it is not possible to infer the original state from the outcome.

(i)

```
// int x;
if ( x > 10 ) {
  x = x - 1;
}
```

(ii)

```
// int x;
if ( x > 10 ) {
  x = x + 1;
}
```

Figure 1: Exam's reversibility question.

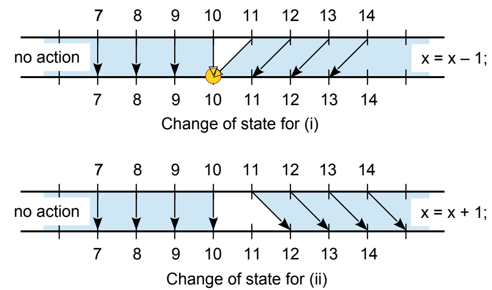

Figure 2: Change of state for the two code fragments

the *if* path, when the condition is true and the *else* or empty path when the condition is false.

Figure 2 illustrates how these two paths "merge" or "separate". When the two paths don't overlap, as in conditional statement (ii) the code is reversible. To reverse the code we may need to adjust the condition so that it reflects the change of state of its variables. When the two paths overlap, as in conditional statement (i) the code is not reversible for that set of values, for example for the value 10 which has two orginal states: 10 and 11. Thus, the statement is not reversible as a whole.

Note it may be possible to reverse the effect of the conditional statement for the range of values that do not overlap. For example, if x's current value is 15 we could deduce or calculate its original value to be 16. Similarly, if x's current value is 8 we could deduce its original value was still 8. Thus, to identify reversibility we need to focus on the borderline cases.

3.2 Data collection

We collected the exam answers from two first year CS undergraduate cohorts at universities located in different countries. Each exam paper answer was anonymised and scanned prior to analysis.

Cohort UN1, comprising 73 students, have taken a full year programming course (24 weeks: overall about 80 hours of lessons + 60 hours of Lab and classroom exercises) based on a functional-first approach with Scheme (functional programming) and then Java (imperative programming and basic notions of object-orientation). The reversibility task included 6 questions—only the first two are anlysed here—and was the first one of four exercises on the final exam; it was worth 25% of the marks.

Cohort UN2, comprising 155 students, were taking a 12-week semester course focused on Object Oriented Programming in C++. The exercise was part of the fourth question in the exam and was worth 5% on their exam. UN2 is a mixed cohort in terms of programming experience: 56% are following it after a 12-week introductory programming course, thus having a full year of instruction; 28% of them taking this as their first programming course at undergraduate level (assuming to have basic programming skills) and 15% doing this course 3 or more semesters later that their first introductory programming course, due to degree transfers or late start of a double degree. We should also note 17% of students are repeating this course as it is core to their degrees.

3.3 Analysis

Firstly, a quantitative analysis was carried out to classify each student selection (reversible or not reversible), plus their explanation or code as correct/incorrect. This measures how many students have mastered the concept tested.

However, it is also very important to understand the mental models or misconceptions that caused students to fail. Thus, in order to gain insight into their level of mastery of the task, we conducted a qualitative analysis based on their two combined answers. The SOLO taxonomy [1] has been used [9, 14, 17] to classify student answers to coding (read/write) questions. As Shuhidan's concluded "[...] we recommend the SOLO Taxonomy, to measure the novices' understanding of the particular concepts tested. The SOLO Taxonomy provides a means of evaluating cognitive or mental models, to see if the novices are able to make connections between what they have learnt, if any exist." In other words, besides classifying a task at a given level, it helps to observe the progress made by each student towards that top level.

We considered the proposed task to be at the Relational level for novice programmers; thus, we mapped each student's answer into SOLO levels in accordance with the criteria listed in Table 1.

Two of the researchers assessed all students' answers and performed a deductive content analysis [10] based on the four SOLO categories. Overall, the rate of agreement was about 93%. By considering each category separately, the percentage of agreement varied from about 95% (*multistructural* level) to almost 99% (*prestructural* level). Cohen's Kappa measure of the intercoder reliability was in the range 0.83–0.96. Eventually, further discussion led to the final classification summarized in Table 3, to be discussed in the next section.

Table 1: SOLO Classification of answers

SOLO Level	Conditional statement and state
Prestructural (P)	poor answer that shows lack of understanding of the task, poor programing skills or misinterpretation of what reversiblity means.
Unistructural (U)	partial understanding of reversibility; student makes a simplistic attempt to reverse the code inside the if.
Multistructural (M)	clear understanding of the reversibility task, but one of the two exercises fail in some way, such as being able to correctly reverse the second conditional statement, but failing to identify or explain the overlap between the two paths.
Relational (R)	concept well understood and applied correctly to both questions: they explain the overlap for (i) and provide code to reverse (ii).

Table 2: Student answers - Is this code reversible?

	(i) not reversible	explains overlap	(ii) reversible	correct code
UN1	49.3%	39.7%	97.3%	89.0%
UN2	49.0%	27.7%	83.2%	51.0%

Table 3: Classification of students' answers.

SOLO Level	UN1	UN2
Relational	38.4%	23.2%
Multistructural	43.8%	34.8%
Unistructural	12.3%	15.5%
Prestructural	5.5%	16.8%
Empty	0.0%	9.7%
SOLO mean	3.15	2.45

4 RESULTS

4.1 Quantitative analysis

Table 2 shows the student responses for the task shown in Figure 1. The percentages of students that provided the correct answer for the first code fragment is similar, close to 50% in both cohorts, although UN1 student are more precise at explaining the overlap that is the cause of the code not being reversible. Note that the previous study [5], using a small sample reported only 20% of them identify code (i) as not reversible. Thus, there is a significant improvement on both cohorts after 12/24 weeks of instruction. This indicates reasoning about reversibility is within the reach of novice programmers.

4.2 Qualitative analysis

Next, we look at the outcome of applying the SOLO taxonomy criteria outlined in subsection 3.3 to classify the answers. The results are summarized in table 3, where the means are obtained by assigning a value in the range 1–4 to each SOLO level (1 = Prestructural and 4 = Relational), or 0 if the answer is empty or without justification, and then averaging over the assigned values.

Relational answers amount to 28.1% of the rated sheets overall. They were provided by students that answered both questions correctly and accurately. Figure 3 shows an example of a well-formulated answer at the Relational level. As educators, however, we are particularly interested in identifying common patterns and misconceptions that prevent students from reaching the relational level. Thus, in the rest of this section we will describe the range of answers, from the lowest level up.

i) No. if x = 10 at the end, was it because it was always 10, or because it was 11-1? It cannot be reversed

ii) Yes.
```
if (x > 11) {
  x=x-1;
}
```

Figure 3: Example pertaining to the Relational level

4.2.1 Prestructural answers. Prestructural answers (13.2% overall) have poor code or vague explanations and mostly indicate lack of preparation for a programming exam. However, some of them illustrate a poor understanding of the reversibility task. The first example of figure 4 shows how a student approached the task by flipping the condition "x > 10" to "x <10", which indicates a very superficial grasping of reversibility. A few students attempted to reverse the action inside the original code as shown in the middle excerpt of Figure 4, and a couple of them tried to prove it can be reversed for all values by including a for loop in their code as sown in the last example of figure 4. The 3 examples represent students that have mastered the C++ syntax, but have a very superficial grasp of the corresponding semantics.

4.2.2 Unistructural answers. Students at this SOLO level (14.5%) have a naive view of reversibility and their answers reflect their belief that they can just focus on the block inside the conditional statement and ignore the control flow. Figure 5 shows a few variations of this theme. Students may just report the command they modify, or they will copy the rest of the if conditional as provided. Most of them answer "Yes" to both question as in three or the four sample answers shown in Figure 5. The last answer says "No" to the first question, but the logic is flawed as there is no risk of underflow. Moreover, its second answer fits the pattern describe above.

4.2.3 Multistructural answers. Answers at this level (37.7%) are above the naive pattern but still are incorrect in some way. Most of them fall into two patterns: in the first case the students think both conditionals are reversible, and in the second case they identify the first conditional as not reversible. The first pattern (answer Yes-Yes, 23.2% of the sheets) is shown in the leftmost example of figure 6. They change the assignment, as seen in the unistructural level, but they also adjust the condition accordingly. The answer for (ii) is correct, but that for (i) misses the overlap.

The second pattern (answer No-Yes, 11.0%) is shown by the middle example in figure 6. In this case, the students have realised that the typical naive solution found at the unistructural level does not work when the original value of x was 11. So, their reasoning is partially correct, but they fail to identify the overlap as the key reason for not being reversible. A longer description of this kind of reasoning for item (i) is provided in the last example of figure 6. This group of students get the correct No-Yes choices, but they do not adjust the condition of the second conditional. Note the code they wrote will work, as if the original value was 11, it will now be 12 and therefore greater than 10; thus, it is not clear from the exam papers if they are aware of this or not.

There remain about 3.5% of the answers which do not match to the two patterns outlined above. In a couple of them, for example, we can also find non-legitimate pieces of code introducing a new variable to save the initial value of x in order to restore it directly, after the execution of the given conditional statement.

5 DISCUSSION

We now revisit the research questions in light of the results just presented. To answer our first question, "Can novice programmers reverse a conditional statement?" we need to look at how many students succeeded in reversing part (ii).

Based on the quantitative data, 63% of students were able to reverse the code correctly. However, this figure considers all variations of code that return to the original state, including those using the original condition "x > 10". On the other hand, based on the qualitative analysis, if we take into account the students at the multistructural level that adjusted the condition to reflect the change of state, plus those at the relational level, then the answer is 51.3%. Note the unistructural students could not use their knowledge of the conditional semantics to complete the task successfully.

In regards to the second question, "How comprehensively can novice programmers envisage the impact of conditionals in state variables?" we have addressed this issue using the SOLO taxonomy. As both the code and the answers implied by the assigned task are brief, the key differences are in the students' awareness of the two alternate paths of execution, and in the quality of their explanations.

In SOLO scale terms, a SOLO mean of 2.68 across the two cohorts is similar to or above that reported in other studies that use this taxonomy to evaluate first year undergraduate performance. More specifically Sheard [13] reported a mean score of 2.15 for a reading skill task that required students to explain the effect of code that contains a loop over an array (the code checks if the array is in order). Shuhidan's evaluation [14] of a simple vector iteration resulted in a mean score of 2.51. Note this reasoning task was new in the exam, as well as the concept of reversibility which was not covered previously during instruction. Thus, we had lower expectations compared with other coding exercises which they are familiar with, as they have received instruction and feedback during the semester(s). In particular, the UN1 cohort exceeded expectations with a SOLO mean score of 3.15 for an unfamiliar task.

Finally, in terms of understanding reversibility, both student cohorts show much better performance compared to the results reported by Teague and Lister [16] for a range of cohorts with 12/24 weeks of class instruction. Their best class had only 32% correct answers for reversing a vector operation inside the loop; on average they provide 25% correct answers for a coding-only exercise.

Limitations. When students decide the code is reversible, we are not aware of their reasoning as we only asked them to provide the code. Thus, for the second task they may get it right and still have a shallow understanding of reversibility.

A second limitation for the UN2 cohort was the time constraints and the low percentage of marks (5% of final result) for the question that influenced students to skip over it as shown by its percentage of empty answers. Other students may have rushed on their answers to put their time onto the more familiar formatted questions.

Figure 4: Examples of Prestructural answers

Figure 5: Examples of Unistructural answers

Implications for educators. We want our students to become both self-diected learners, and to develop higher-order thinking skills needed to reason about programs as a whole. This work demonstrates that the concept of reversibility can be used to explore program comprehension using low-ceiling tasks such as variations of the question used in this study.

We also need to investigate what types of instruction is required to make students aware they need to consider the potential overlap between the two execution paths of the conditional statement so they will be able to understand the range of possible state changes caused by the conditional statement as a whole.

When giving feedback about coding exercises, we should focus not only on the lines the students did wrong but give more emphasis to their reasoning or lack of about program behaviour. In other words, we should look for opportunities like this exercise for students to become aware and improve their higher-order thinking skills. This will be useful for those working at levels closer to the correct answer.

A possible way to help students to explore their reasoning is to provide peer review activities that ask them to reason about reversing or debugging, again with a focus on explaining instead of correcting code.

6 CONCLUSIONS

Although most novice programmers understand the semantics of conditional statements and are able to read and explain them, recent work has indicated students may fail to reason comprehensively and understand as a whole the impact that a conditional statement can have on the state of the variable it modifies.

Two first year undergraduate cohorts from different countries were asked to complete a reasoning and coding task on a simple *if*

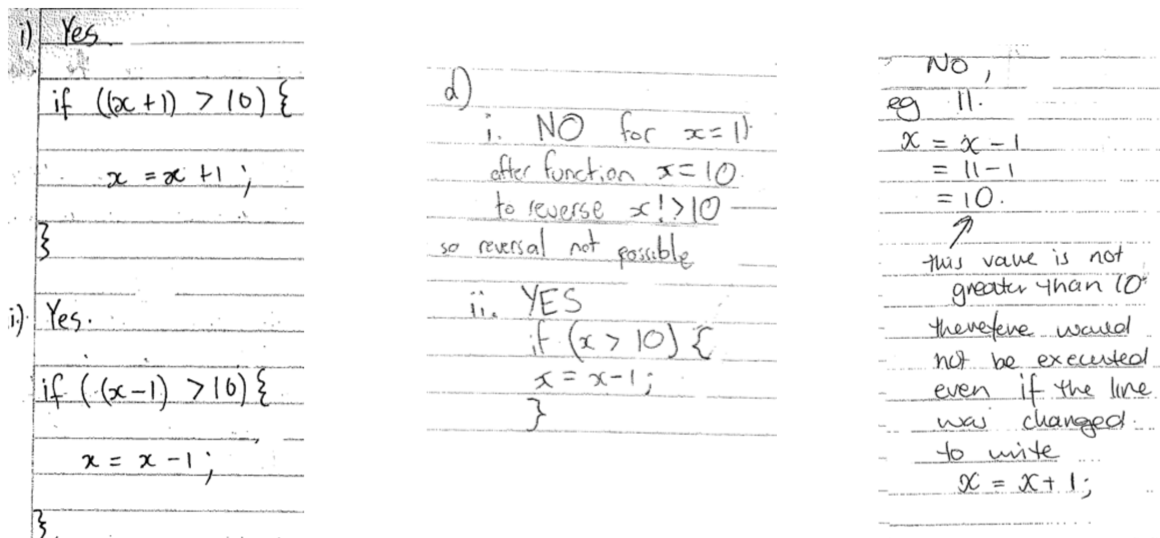

Figure 6: Examples of Multistructural answers

statement as part of their final exam paper. Reversibility was introduced through examples, which required students to understand this abstract concept and to envisage how to apply it to small code statements.

This study has evaluated in depth the students' ability to reason about and reverse conditional statements. In spite of the minimal dataflow overlap, 28% of students showed that they understood how multiple flows (decrementing a variable x versus doing nothing) can result in the same final value. In addition, more than half of the participants involved in our investigation—i.e. at least 51%—appear to be definitely able to devise correct code for undoing a *reversible* conditional statement. On the other hand, the answers classified at the lowest prestructural level of the SOLO taxonomy, indicating serious lack of understanding of the reversibility concept and/or of the semantics of very basic programming constructs, amount to only 13% overall.

Finally, as a future direction of work, we plan to extend our investigation on reversibility to address different code fragments, namely conditional with both branches made explicit (*if-else* statements) and simple loops, as well as to include other student cohorts.

REFERENCES

[1] J. B. Biggs and K. F Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* Academic Press, New York, NY, USA.

[2] R. Bornat, S. Dehnadi, and D. Barton. 2012. Observing Mental Models in Novice Programmers. In *Proc. 24th Annual Workshop of the Psychology of Programming Interest Group (ACE '12).* 77–86. http://www.ppig.org/papers/24/8.Observing_mental_models-Richard%20Bornat.pdf

[3] Ibrahim Cetin. 2015. Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective. *Canadian Journal of Science, Mathematics and Technology Education* 15, 2 (Feb. 2015), 155–170. https://doi.org/10.1080/14926156.2015.1014075

[4] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14).* ACM, New York, NY, USA, 695–700. https://doi.org/10.1145/2538862.2538966

[5] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. 2017. On the Ability to Reason About Program Behaviour: A Think-Aloud Study. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17).* ACM, New York, NY, USA, 305–310. https://doi.org/10.1145/3059009.3059036

[6] FJ King, L Goodson, and F Rohani. 1998. Higher order thinking skills: Definitions, strategies, assessment. *A publication of the Educational Services Program (now known as the Center for Advancement of Learning and Assessment). Retrieved December* 4 (1998), 2014. http://www.cala.fsu.edu/files/higher_order_thinking_skills.pdf

[7] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), 9–18.

[8] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. *SIGCSE Bull.* 38, 3 (June 2006), 118–122. https://doi.org/10.1145/1140123.1140157

[9] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proc. 4th Int. Workshop on Computing Education Research (ICER '08).* ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[10] Philipp Mayring. 2014. *Qualitative content analysis: theoretical foundation, basic procedures and software solution.* Klagenfurt.

[11] Nancy Pennington. 1987. Empirical Studies of Programmers: Second Workshop. Ablex Publishing Corp., Norwood, NJ, USA, Chapter Comprehension Strategies in Programming, 100–113. http://dl.acm.org/citation.cfm?id=54968.54975

[12] Noa Ragonis and Mordechai Ben-Ari. 2005. On Understanding the Statics and Dynamics of Object-oriented Programs. *SIGCSE Bull.* 37, 1 (Feb. 2005), 226–230. https://doi.org/10.1145/1047124.1047425

[13] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, and Errold Thompsom. 2008. Going SOLO to assess novice programmers. In *ITICSE'08.* 209–213. https://doi.org/10.1145/1597849.1384328

[14] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A Taxonomic Study of Novice Programming Summative Assessment. In *Proc. 11th Australasian Conf. on Computing Education - Volume 95 (ACE '09).* Australian Computer Society, Inc., 147–156. http://dl.acm.org/citation.cfm?id=1862712.1862734

[15] Peter Sutherland. 1999. The application of Piagetian and Neo-Piagetian ideas to further and higher education. *International Journal of Lifelong Education* 18, 4 (1999), 286–294. https://doi.org/10.1080/026013799293702 arXiv:http://dx.doi.org/10.1080/026013799293702

[16] Donna Teague and Raymond Lister. 2014. Programming : Reading , Writing And Reversing. *Proc. 19th ACM Conf. Innov. Technol. Comput. Sci. Educ. - ITiCSE '14* (2014), 285–290. https://doi.org/10.1145/2591708.2591712

[17] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. *Conferences in Research and Practice in Information Technology Series* 114 (2011), 37–45.