Measuring Programming Knowledge in a Research Context

Kristina von Hausswolff
Uppsala University
Department of Information Technology
Uppsala, Sweden
kristina.von.hausswolff@it.uu.se

Anna Eckerdal
Uppsala University
Department of Information Technology
Uppsala, Sweden
anna.eckerdal@it.uu.se

***Abstract*—There is a continued need for basic research on how beginners learn programming across different contexts. In research aiming at measuring effects of learning, ways of measuring programming knowledge are needed. The aim of this paper is to describe a process of establishing validity of a context-specific assessment tool for novice programming. Our case is an ongoing research project that investigates the effects of and mechanisms behind learning computing programming from hands-on experiences. For that context, we have constructed and validated an assessment tool i.e., a programming knowledge test with a connected scoring rubric based on an introductory three-hour teaching session. We relied on three ways of arguing for how our assessment of knowledge is considered valid in this process: previous research, an empirical sample, and intersubjective knowledge judgments. The main result of our study is the test with the scoring rubric and our arguments for its validity. We also present some tentative findings from the research project on the role of hands-on in learning programming. Both the area of hands-on, as well as the context-specific process, seem to be fruitful topics of future research, particularly combined with a pragmatic action and communication theory.**

I. Introduction

Basic knowledge of computer programming is now considered relevant for all citizens. Similar to many other countries, Sweden's National Agency for Education has proposed a new curriculum that explicitly introduces programming and strengthens digital competencies. With new curricula that explicitly introduce programming into schools, there is an increasing need for basic research about how beginners acquire relevant practical and theoretical knowledge in the subject domain. In research aiming at measuring effects of learning, e.g., in experiments, quasi-experiments, or in evaluating an intervention, assessment tools that measure programming knowledge are needed. Standardized tools to measure knowledge in computer science education are rare due to the subject novelty in schools. Even if a standard test is available, it may not fit the particular context, including the particular content. This is the case in our research project and a context-specific assessment tool had to be developed.

In this paper we describe a process of developing such a context-specific assessment tool. Our case is an ongoing research project in Sweden and we constructed an assessment tool specific to that context (i.e. a programming knowledge test with a connected scoring rubric). The relevant parts of the larger research project will be described in order to give an understanding of the case-specifics that make out the preconditions for our assessment tool. We have chosen to rely on three ways to argue for how our assessment of knowledge is considered valid in the process of developing our context-specific assessment tool: previous research, an empirical sample, and intersubjective knowledge judgments. The participants in the projects had no or minor previous knowledge of programming. Based on previous research, we thus selected the concepts to go into our lesson plan, namely learning simple syntax combined with sequence and loops.

To establish a valid scoring rubric, three researchers in the field of computer science education, who are also experienced teachers in programming at the university level, evaluated the student answers. These intersubjective knowledge judgments were used to establish interrater reliability, which builds on a Deweyan account of knowledge. The three researchers were involved in a process consisting of three phases of independent grading with incrementally developed scoring rubrics and critical knowledge assessment discussions.

A. Research Context

The ongoing research project, that this study is a part of, investigates the effect of and mechanisms behind learning computing programming from hands-on experiences, at upper secondary school and university (described further in [1]). As a part of this research project we will study novice students learning to program in an experiment setting in a computer lab. At the core of the experiment is a three-hour teaching session were students with no previous programming knowledge work with programming assignments. The teaching session consists of the teacher presenting programming concepts and showing examples of problem solving with programming. The students work in groups of two, a pair programming inspired setting [2]. In pair programming two persons work together to solve a problem at a computer. They alternate between being the "driver", who types the code, and the "navigator", who is responsible for detecting errors and helping to solve the problem. In our setting however, the driver and navigator never change roles.

The reason for this design is that it supports the investigation of hands-on vs. hands-off, in that both students in

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

a pair are involved in the problem solving, but only one is hands-on during the teaching session. We aim to investigate how attention, motivation, stress and long-term memory are affected by hands-on and hands-off learning, respectively. Previous research has suggested that these factors may explain the beneficial effects of hands-on. This experiment will also include functional magnetic resonance imaging (fMRI) of participants' brains shortly after the teaching session. All data will be correlated with the results from the programming knowledge test to evaluate what role each factor plays. The project is thus greatly dependent on a validated assessment tool.

B. Aim and Research Question

The aim of this paper is to describe a process of establishing validity of a context-specific test with a connected scoring rubric for novice programming. The overall research question is thus: How can validity of a context-specific knowledge test for novice programming be established?

II. Background

A. Choice of content

In our teaching session we introduce simple syntax, sequence, and while-loops to the students. We refer to these as three categories, where sequence and while-loops are concepts. Other categories could have been chosen, but simple code writing and reading tasks require knowledge of basic syntax and sequence. In addition we decided that the while loop concept would be useful since it opens for many different exercises. These categories are also identified as central in many common CS1 text books and curricula documents [3]–[5] as well as in previous research.

Soloway [6] and Goldman et al. [7] identified loop as an important but problematic concept to learn, and Kaczmarczyk et al. [4] found "a number of misconceptions all related to an inability to properly understand the process of while loop functioning" among novice programmers. The authors further found that "[s]tudents cannot trace code linearly" (p. 108). The skill *linear tracing* relates to the concept sequence which is one of our key concepts.

In their literature review investigating learning trajectories of basic concepts in K-8 programming education, Rich et al. [8] focused on sequence, repetition and conditional. Concepts for beginners are suitable for our research on learning objectives in the first thee hours of learning programming. Two of the concepts are interesting for our purposes: sequence and repetition. Rich et al. showed a number of learning objectives for each concept which they had divided into categories of beginning, intermediate, and advanced (p. 187). For the concept sequence, our teaching session matches the authors' listed objectives, namely that the order of instruction is important for the output; and that new instructions can affect when and which instructions are executed. For the repetition trajectory, we have diverged in two aspects. First, we focused on the while loop instead of discussing several kinds of repetition. We included all other learning objectives at the beginning and intermediate level. Second, we omitted the learning objective about nested loops, which were the only one classified as advanced.

B. Assessment of programming knowledge

Previous research has discussed how to best assess programming students' knowledge. Luxton-Reilly et al. [9] examined 417 exam questions including code, developed for first programming courses. Their analysis of the questions revealed that seemingly straightforward questions contain multiple components. Assessing using such complex questions means that students can fail for many reasons, while being prevented from showing concepts they do know. The authors argued for a wider use of single-concept assessment questions. The results of Luxton-Reilly et al. indicated that we should embrace programming knowledge as situated in a specific context that depends on the learned programming language. Consequently, the syntax of that language would be part of programming knowledge together with knowing specific concepts.

Similarly, Robins [10] claimed that the compound and intertwined nature of the programming subject is a reason for students' problem to learn to program. Petersen et al. [11], following Lister [12], also argued that the way that programming is assessed could cause the high failure rate in CS1 classes, together with the many intertwined concepts. In a review of fifteen exams from fourteen North American institutions, Petersen et al. examined assessments of both skills and concepts. They found that writing code questions where commonly used to assess programming skills, e.g., the skill to write code (54% of the questions), and also short answer or multiple-choice questions to assess reading code. Petersen et al. concluded that both writing code and short answers involved on average seven concepts; three for multiple-choice questions. As a result, they suggested to use more multiple-choice questions that tests only one or two concepts at a time.

C. Validity and Interrater Reliability

Different ways to validate introductory programming tests have been investigated in previous research. Tew [3] developed a validated language independent instrument to measure CS1 conceptual knowledge. To this end she performed document analysis of common CS1 textbooks, and studied the current ACM/IEEE curriculum guidelines [5]. A 2016 replication study [13] extended and further validated Tew's instrument. However, those instruments were unsuitable for our purposes as they differ from our case in three important ways. First, they were designed to test knowledge after a whole semester of teaching, second, they were language-independent, and third, they used only multiple-choice questions. Bloom's taxonomy has been applied to programming tasks and tests [14], and Kasto [15] developed ways to measure the level of difficulty of code writing tasks based on the SOLO taxonomy [16] and commonly used software metrics. The limitation of the content in our test makes however both Blooms' and the SOLO taxonomy difficult to use for us.

Due to the lack of validated tests for CS knowledge in general and programming specifically [17], more research into validation of tests for different purposes are needed. The recent development of and interest in concept inventories in CS addresses one part of this need [18]. When the need is highly dependent on a specific context as in our project, then the process of creating and validating a test for that context is another way of tackling the problem of measuring programing knowledge.

Inspired by Parker et al. [13] we will discuss validity in terms of construct validity and content validity. The authors present a study similar to ours in that they investigate the validity of a knowledge test. They write:

> *Construct validity* considers the extent to which performance on an assessment can be interpreted in terms of one or more constructs. Construct validity is thus intended to show that the test measures what it is intended to measure. *Content validity* considers the extent to which assessment questions provide an adequate and appropriate sample of the domain tasks. (p. 97, italics in original).

Parker et al. write that the two types of validity together "provide evidence that an assessment is working as intended" (p. 96).

In the present study, content validity was established by a thorough selection of which main concepts, i.e. *sequence* and *loop*, to introduce in the teaching material. Beside the concepts a category of knowledge connecting to the practical writing in the Java language were included, i.e. *syntax*. The selection was further detailed and included parts like the tokens curly bracket and semicolon, which belong to the category *syntax*. All parts were listed and checked by both authors. From this list we constructed the test questions.

Construct validity was established in that the categories were combined with the skills *read* and *write* code, resulting in two types of questions: code-writing questions and non-writing questions. Non-writing here means that the answer to the questions did not include code-writing or sentences in a natural language. Construct validity was strengthened by the fact that the different categories and question types were informed by and grounded in aforementioned previous research and subsequently tested by the commonly used measurement Crombach's alpha [19].

The empirical sample described in this paper then provided us with information on the level of difficulty of each question so that we could fine-tune the final test after we had developed the scoring rubric and graded the student answers, see Section III.

We will here discuss reliability in terms of interrater reliability. Stemler [20] argues that interrater reliability could be denoted by different measurements. Two of those are *consensus estimates* and *consistency estimates* (p. 1).

Consensus estimates are based on the assumption that different raters should be able to agree exactly on how to use the rating scale when rating student answers. We will use consensus estimate to measure two researchers' independent ratings on the same student answers at different times in the process.

Consistency estimates are based on the assumption that exact agreement is not necessary between raters as long as the raters are consistent within the scale applied to categories, score or applying a scoring rubric [20, p. 3]. We will use consistency estimates for comparing the scoring between different scoring rubrics at different times in the process of constructing a scoring rubric.

D. Theory of Learning and Intersubjective Knowledge

Our theoretical standpoint is a pragmatic view of knowledge and learning. This research is focused on hands-on experience in learning programming and has a strong emphasis on practice in acquiring both knowledge and skills. The embrace of practice in education is not a new phenomenon. The pragmatic thinker and educationalist John Dewey argues practical action as necessary part of acquiring knowledge, in his pragmatic action and communication theory [21]–[23]. As described by Biesta and Burbles [24], a pragmatic view on learning is a process of inquiry and builds on an ontology of transactional realism and an intersubjective theory of knowledge. The learner has a transaction with their environment in a process of inquiry. Actions together with already established knowledge inform the learner and intersubjective knowledge is created through communication. Dewey's intersubjective theory of knowledge holds that knowledge is not objective nor is it completely subjective. The intersubjectivity of knowledge relies both on already established knowledge (in an informed community) and the communication with peers in that informed community.

In this paper the pragmatic theory of intersubjective knowledge plays an important part in arguing for the validity of the constructed tool (i.e. a programming knowledge test with a connected scoring rubric). We argue that the tool measures knowledge in programming in an intersubjective way that takes the established community of educators in programming into account.

Previous research has pointed to the beneficial effects of hands-on in programming education research [25], [26]. The important role of hands-on in computing education in general is highlighted in the considerable body of research on the role of practice, both as means to reach learning goals, and as a goal of itself [27], [28]. There seems to be consensus among students as well as teachers that hands-on is important [29]. Lahtinen et al. [29] found in a survey on programming education with answers from 559 students and 34 teachers that

> the biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them. [...] both students and teachers agreed that the practical learning situations were the most useful (p. 17).

Not only the application of theoretical knowledge is important, the learning goes both ways—also from practice to conceptual knowledge, as shown in [30]. A spiral approach to teaching programing has been advocated for a long time [31] where

learning is built up by moving between syntax programming and semantics in terms of different concepts. It builds the conceptual understanding by visiting the same concept several times in greater depths. This is also one of the ideas behind our design of the teaching session.

III. Developing the Context-specific Knowledge test and scoring rubric

The development of the context-specific knowledge test and scoring rubric was done over three phases, described in detail in the following sections. Three researchers/programming teachers were involved, denoted Researcher A, B, and C.

Phase I: Researchers A and B developed the lesson plan. Researchers B and C constructed the test.

Phase II: Researcher C conducted the fieldwork of the empirical data sampling, while Researchers B and C did the initial evaluation of the student answers and constructed the first scoring rubric in that process.

Phase III: Two parallel processes were conducted: One process consisted of Researchers A and C establishing the interrater reliability. The other consisted of all three researchers establishing the scoring rubric and ensuring the intersubjective knowledge measurement of the test.

A. PHASE I

The teaching was conducted in Swedish as was the knowledge test. Necessary parts of the test were translated into English for this paper.

1) The teaching session: The goal of the first part of the teaching sessions (1.5 h) was to initiate understanding of sequence of instructions through the Java language and text output, to enable students to write code that compiles and gives the desired outcome. This introduction necessarily included a focus on Java syntax e.g., declaration of variables, variables types, curly bracket, semi-colon at the end of each statement, and so forth. The general understanding of how to give instructions in sequence was taught simultaneously with the Java syntax.

The second part (1.5 h) relied on the learned understanding of syntax and sequence but extended this knowledge to loop statements (only while statements). In addition to introducing loops as a concept the statements were shown in a graphic setting letting the program control the movement of a turtle.

We used a spiral approach to our teaching, as advocated by [31]. Explanations and exercises were interleaved, and concepts were introduced with gradually more depth and repeatedly, and with variation [32].

2) The test: The test was developed in relation to what was taught during the teaching session, namely simple syntax and the concepts sequence and loop. Results from previous research [9], [11], [33] shows that simple Java syntax as well as more semantic concepts causes difficulties when answering programing questions. This suggests that at least some questions in our test should assess concepts and syntax independently, which is in line with [11]. The code-writing questions were analyzed into parts belonging to different categories (for more detail see Section IV-B).

Another variable to consider when constructing the test was the form of the questions. Two main considerations were taken into account. The first one was that the students wrote programs during the teaching session and that this should be reflected in how the knowledge was tested. In three of a total of eight questions, students were instructed to write code on paper displaying a solution to a programming task (code-writing questions). The code-writing questions required an answer of about ten lines of code, a compound answer that includes several concepts and syntax. Some of the non-writing questions were constructed to test parts of the content independently.

The second consideration is the need of resemblance between questions in the test and questions for our planned fMRI setting, i.e. yes and no questions. Five of the eight test questions were either multiple-choice questions or questions were the student should give the output of a displayed program (non-writing questions). These types of questions could easily be altered to variations of yes and no questions and thus prepare the students for the fMRI test. Figure 1 illustrates the final structure of the test.

B. PHASE II

The second phase was the empirical data sampling.

1) Experimental design: The purpose of this empirical data sampling was to validate the knowledge test in a setting that was as naturalistic as possible. Three hours of teaching was conducted, after which the students took the knowledge test. The experiment design was an between-groups study (hands-on versus hands-off).

2) Participants: One of the authors contacted three upper secondary schools asking if students in the natural science program could participate in our study, the short introductory Java course including the teaching session with a subsequent knowledge test. Three classes participated, two of them in one school where the participating students were in their first year of the study program, age 16 or 17. The third class was in another school with students in the second or third year, age 18 or 19.

After the first two sessions, the test was evaluated and some minor errors corrected. The order of the questions was altered so that it reflected the level of difficulty. The third student group was taught a month later. The participants were slightly older and the setting was more focused. 75 students participated in the three sessions altogether, of which 15 had previous programming experience. A pre-condition for inclusion in the empirical sample is that the participants had never programmed before. Therefore, the 15 students with previous experience were excluded.

3) The teaching session: In the pre-planned and structured teaching session the teacher explained syntax and concepts and showed examples on a screen. The students listened and performed practical exercises on the computer. The students worked in pairs, about half of the students hands-on and the

Fig. 1. Final structure of the test. The width of the area of each question corresponds to its maximum score.

other half hands-off. The assignment of hands-on and hands-off was randomized within the pairs. The students themselves selected their partner to work with. Even though we tried to get even pairs and instructed the students to be either hands-on or hands-off some of the students ended up working alone resulting in slightly more students in the hands-on group than the hands-off group. The programming language used was Java with DrJava development environment [34]. Directly after the teaching session, the students took the test individually measuring their knowledge of programming.

C. PHASE III

The final phase involved the process of ensuring the inter-rater reliability and validity of the test. To ensure that the three researchers reached a rating agreement, we used a procedure of independent ranking of difficulty of each question, the independent construction of different scoring rubrics, and individual markings with several scoring rubrics.

In a parallel process, we aimed at validating the test as a knowledge assessment tool that allows rating on an interval scale. This last process included individual markings, independent judgments of the relative weight of the questions in relation to each other, and a holistic comparison between some of the student answers.

We selected 16 student answers for a more detailed examination. They were all answers from the students with no prior experience of programming from the third session in the empirical study. The third session was selected as the setting and the age of the participants closer resembled the planned experiment. All of the activities in Phase III were conducted on these 16 student answers.

1) Agreement of the consensus estimate: Developing the final detailed scoring rubric R4 included constructing several rubrics independently, scoring with different rubrics, and discussions between researchers. Discussions took place during three meetings approximately one month apart, with independent work before and after each one. We label these meetings m_1, m_2, and m_3. The time before m_1 is labeled t_0, the time after m_1 is t_1, and so on. Researchers A and C both took part in the meetings as well as in the individual work in between. Researcher B took part in the two last meetings (m_2, m_3). Four rubrics, R1–R4, were developed, R4 being the final one.

Initially (t_0), Researchers A and C created two scoring rubrics independent of each other (R1 and R3). Both researchers then used one of them, R1, and independently scored 16 anonymous student answers. This was done to evaluate different ways of judging programming knowledge. The scoring was later compared and the differences in scoring were discussed question by question. Their ratings had a high agreement ratio considering that scores could range from 0 to 55 and only exact agreement was counted. The agreement ratio is higher still if the exactness demand is loosened to allow a difference of at most 1 point as suggested by Stemler [20]. During meeting m_1, rating errors were discovered and judgments were discussed, resulting in an updated scoring for both researchers and a higher percent agreement (see Table I). The minor disagreements mostly concerned the code-writing questions and helped formulate the scoring rubric in a more exact form (R2).

The other initial rubric (R3) was then used in a similar way—first scoring independently and then comparing. Both researchers A and C marked the 16 student answers again using this scoring rubric (Rubric R3) with a high percent agreement (shown in Table I). The reliability of the scoring rubric was tested simultaneously.

TABLE I
LEVELS OF AGREEMENT: STUDENT ANSWERS RATED AT VARIOUS TIMES IN THE SCORING RUBRIC DEVELOPMENT.

Time	Scoring rubric	Agreement ratio	Agreement ratio (difference $\leq$ 1 point)
t_0	Rubric R1 (0–55 points)	31 %	94 %
t_1	Rubric R2 (0–55 points)	56 %	100 %
t_1	Rubric R3 (0–34 points)	56 %	88 %

t_0: Before the first meeting. t_1: After the first meeting

2) Validation of the scoring rubric and consistency estimates: To establish the weight between the questions and within different parts of one question, all three researchers met (m_2) and discussed the relative difficulties. Examples from the 16 student answers were used in the discussion. The initial scoring of the 60 student answers using Rubric R2 was also used to evaluate the difficulty of the questions. The average solution rate ranged from 13 % to 77 % over the different questions. The question that got 77 % average solution rate was considered the easiest, while the question getting 13 % solving rate was considered the most difficult. After this meeting (t_2) a further improved scoring rubric was constructed (Rubric R4). It had a range of 0–44 points and could, with our calibration of knowledge judgments, now measure programming knowledge in our specific context on an interval scale.

For a final validation of the scoring rubric, an additional meeting was arranged (m_3). All three researchers were instructed to rate pairs or triplets of student answers that had come close in the scoring, without using the scoring rubric, and argue for their rating. In this final meeting a high degree of agreement was found but an issue with the scoring rubric was detected. In the aim of getting precise instructions in the scoring rubric, the holistic quality of an answer was sometimes lost in one or two of the code-writing questions. It had

become fairly easy to get points for demonstrating fragmentary knowledge of programming even though the answer in a holistic sense was of poor quality. A restriction on getting any points at all on two of the questions was thus added. The non-parametric statistical test we planned to use, Wilcoxon significance test, relies on the order of the data. Because of that we wanted to ensure that we got the order in accordance to the researchers' intersubjective judgment. We thus developed a method of separating student answer results with the same score from each other, by judging some questions as more difficult. The final scoring rubric, with this restriction, we still denote Rubric R4.

Correlation analysis confirms a high level of consistency between the different rating rubrics, shown in Table II. Correlation is measured with Spearman's rho as the data does not follow a normal distribution. Stemler [20] suggests this as an appropriate measurement for grading with different scales but with the same underpinning construct. We thus used this measurement to control the effect of using different scoring rubrics while grading the same student answers.

TABLE II
CORRELATIONS OF TEST SCORES JUDGED ACCORDING TO SCORING RUBRICS AT VARIOUS TIMES IN THE DEVELOPMENT.

	Rubric R2/t_1 Researcher A	Rubric R3/t_2 Researcher B	Rubric R4/t_3 Researchers A,B,C
Rubric R1/t_1 Researcher B	0.975**	0.979**	0.976**
Rubric R2/t_1 Researcher A		0.989**	0.974**
Rubric R3/t_2 Researcher B			0.980**

t_1: After meeting one. t_2: After meeting two. t_3: After meeting three.
** Correlation is significant at the 0.01 level (2-tailed), N=16.

IV. RESULTS

The main result of the study reported in this paper is the context-specific assessment tool, namely the test with the scoring rubric, and the development process described in this paper. In this section we will discuss some results and constrains identified.

A. The test and the scoring rubric

This test is now ready to be used in a controlled experiment setting, with the potential to test hypotheses with the non-parametric Wilcoxon significance test. The reason for using the non-parametric test although the score data is on an interval scale is that we have no theoretical reason to assume that the results follow a normal distribution. O'Boyle & Aguinis [35] argued against the assumption that human performance is normally distributed, based on their study of 198 samples that were remarkably consistent across types of performance measures and time frames. According to [10], there is empirical evidence which contradicts the assumption of a bell curve in novice programming students' performance. Our results confirm the theory in that respect, following a non-normal distribution (Figure 2). While the distribution is different from a bell curve, the two categories (hands-on and hands-off) were shown to have the same distribution (Figure 3), also a precondition to use the Wilcoxon significance test.

1) Returning to validity: The lesson plan and the test rely on established categories (syntax, sequence, and loop). The knowledge is tested with two types of questions: code-writing questions and non-writing questions. In constructing the scoring rubric, both the question type and the category are taken into account. This resulted in finalization of the scoring rubric, as well as consensus about the difficulty of the questions established, through the intersubjective process of the three researchers described above. The final structure of the test is shown in Figure 1, with the maximal score of 44 points and the scoring data on an equidistant interval scale. While the categories are different in some respects, they are not independent of each other. The concept of loop depends on the concept of sequence since a loop consists of a sequence being repeated. Furthermore, to enable the students to learn these abstract concepts, the concrete syntax of the Java language was intertwined. For analytical purposes we measure the three categories separately in the test and can hence use the empirical sample to test the construct validity.

An often-used measure of construct validity, internal consistency, is Cronbachs alpha, which refers to how closely related items are [19]. Cronbachs alpha coefficient was 0.73 for our three categories, which is considered acceptable. This indicates that our categories measure the same underpinning knowledge but also differ in some respects. Cronbachs alpha coefficient was also used to test the sameness of the two question types (code-writing questions and non-writing questions). Here, Cronbachs alpha was 0.87 for the two types of questions, which is considered good and indicates that the two types of questions measure the same type of knowledge. Both these results strengthen the reliability of our constructed test.

2) More about the test: As shown in Figure 1 the code-writing questions (numbers 5–6 and 8) could give a maximum of 22 points, which is the same as the non-writing questions (1–4 and 7). The category scores were divided in this manner: syntax 8 points, sequence 10.5 points, and loops 25.5 points. All the categories where divided equally between code-writing and non-writing questions. The loop category gives about 1.5 times the amount of points compared to the other two categories together. That mirrors the structure of the teaching session. The loop category includes the more difficult questions and to some extent builds on the understanding of both syntax and sequence. The width of the area of each question in Figure 1 shows the distribution of the questions maximum score.

B. Examples of the test questions

To further explain and show concretely what the knowledge test looks like, we will present three of the eight questions but only discuss the scoring in detail for one of the questions, number five. The three questions were selected to show different examples from both the categories and the question types. We present the shorter questions due to the space limitation. Two

questions are non-writing questions and one is a code-writing question. The first question on the test (1A) is an example of the type *non-writing* and category *syntax*.

QUESTION 1 A. Which of the statement(s) a) - l) below is/are correct when inserted in line 3 in the program Write?

```
1 public class Write {
2   public static void main(String[] arg) {
3
4     System.out.println(firstName);
5   }
6 }
```

(a) String firstName = Joe
(b) String firstName = "Joe";
(c) firstName = "Joe"
(d) firstName = Joe
(e) String firstName = "Ann";
(f) String name = "Joe";
(g) name = "Joe"
(h) name = Joe
(i) String name = "Ann"
(j) String name = Joe
(k) firstName = "Ann";
(l) name = Joe;

Listing 1: Example non-writing question: syntax

Students were allowed to select one or many answers to this question which contributed to the difficulty. Selecting both correct answers, (b) and (e), gave 2 p (points), while selecting only one gave 0.5 p. For this question the average solution rate was 61 %. This positions question 1A among the easier in the test, as the average solution rate ranges from 19% to 73%. The average solution rate is calculated by dividing the mean value by the maximum value on that question. This also gave an estimate of the difficulty of each question.

Next is Question 5 of our test—the first and easiest of the type *code-writing* questions and in the category *sequence*. We discuss the scoring in detail to give the reader a feeling for the scoring process.

QUESTION 5. Finish the program MyFigure below so that is gives the following output:

```
My rectangle:
The width is 5
The height is 8
The area is 40
The circumference is 26
```

```
public class MyFigure {
  public static void main(String[] arg) {
    int width = 5;
```

Declare at least one additional variable. Use your variables for calculations and output. Tip: there might exist more than one solution. Tip: print-statements start like this: `System.out.println(`

Listing 2: Example code-writing question: sequence & syntax

This question asked the student to write a piece of code. The question gave 7 p maximum; 4 for syntax and 3 for sequence. The syntax scores were determined as follows: 0.5 p for adding the variable *height* with correct value; 0.75 p for using the semicolon token correctly; 1 p for putting the text correctly in the output; 1 p for using the quotation mark token correctly; and 0.75 p for using the concatenation token correctly. The sequence scores were determined as follows: 1 p for using variables in the calculations; 1.5 p for printing all the values calculated and 0.5 p for using variables, and not numbers, in the print statements. The average solution rate for this question was 51 %. Some divergence between researchers was detected through this process, but they were minor and the scoring rubric was subsequently expanded with details to avoid such differences.

Finally, our seventh question is of type *non-writing*, in category *loop*—and the most difficult of the non-writing questions with an average solution rate of 21%.

QUESTION 7. Which output does the program Count give?

```
public class Count {
  public static void main(String[] arg) {
    int y = 5;
    int x = 1;
    while (x < = 3) {
      System.out.println(x*y);
      x++;
    }
  }
}
```

Listing 3: Example non-writing question: loop

The question gave 7 points in total for the correct answer:

```
5
10
15
```

The scores were determined as follows: The first `5` gives 1.5 p; show a repetition three times gives 1.5 p; including `10` gives 2 p; and including `15` 2 p.

Questions 5 and 7 are examples of questions that, as a whole, are categorized as *sequence* and *loop* respectively but still contain possibilities of getting scores in other categories. In this way we are able to track what kind of knowledge the student displayed when answering the questions, while taking into account the intertwining nature of programming knowledge.

C. Statistics from this study

The final grading scores of all the 60 student answers is shown in Figure 2. Figure 2 also shows that the distribution of the scores do not follow a normal distribution. Through the process of establishing a scoring rubric and evaluating the difficulty of the questions, we argue that the test score is on an ordinal scale and equidistant, which sums up to metric data on a interval scale.

The same distribution is shown in both of the student groups hands-on and hands-off (Figure 3), which is one of the preconditions for using Wilcoxon significance test. The Independent Samples Mann-Whitney U test shows that the distribution of the test scores is the same across the groups hands-on/hands-off (significance level of 0.05).

Using the final grading scores, the Wilcoxon significance test was conducted giving a score of $u = 0.09$ (one tail). This

is not a significant result and the higher score of the hands-on group could thus be explained by chance. The results are however interesting as a preliminary indication that the group of hands-on has a tendency to preform better (Figure 4).

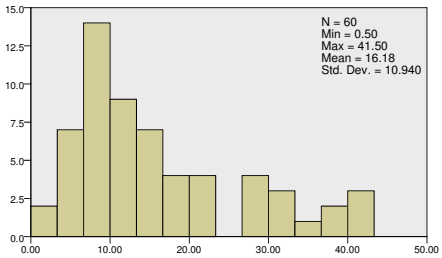

Fig. 2. Histogram of student test scores. The x-axis is the results on the test and the y-axis is the frequency.

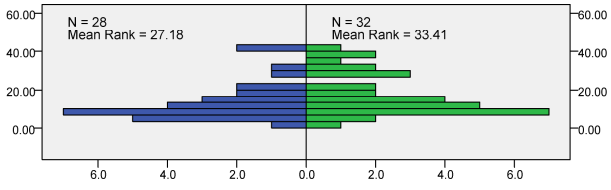

Fig. 3. Distributions of test scores for the hands-off (left) and hands-on (right) groups independently.

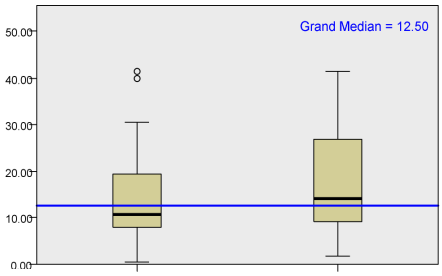

Fig. 4. Median test scores: hands-off (left) and hands-on (right).

D. Other results to be examined in further research

To further explore the results, additional correlation tests were conducted between question category, question type, and hands-on/hands-off. There seems to be a stronger correlation between the outcome of the syntax scoring and hands-on, than between the sequence scoring and hands-on. The loop questions correlated the least with hands-on.

Examining individual patterns of performance, the data show that half of the students perform equally well over all of the categories (30 students), while the other half display patterns of performing better or worse in some category. Seven students show a pattern of preforming better at the code-writing questions, and 16 performed worse at the code-writing questions. There was an equal distribution between hands-on and hands-off and also between female and male students when it comes to the patterns concerning the code-writing questions.

12 students displayed a pattern of preforming better at loop questions, and 15 students displayed a pattern of preforming worse at loop questions. The distribution between male and female was even but the distribution between hands-on and hands-off was not equal when it comes to the results concerning the loop questions. Among the 12 students that preformed better at loops nine were in the hands-off category, and among the 15 students preforming worse at loops 10 were in the hands-on category. As shown above the hands-on category performed better overall. We think these results of individual patterns are interesting and something to look into in future research in the larger research project this study is a part of.

V. Concluding Discussion

We have described a process of creating a valid, context-specific tool for measuring programming knowledge. Our present study, which is part of an ongoing research project investigating the importance of hands-on experience when learning to program, was used as a case for creating a programming knowledge test with connected scoring rubric.

The process includes identifying knowledge content grounded in tradition through previous research, combined with an empirical sample and intersubjective knowledge judgments. *How* these three parts are combined in the described process reflects our view of knowledge in the domain of novice programming.

An important conclusion is that measuring programming knowledge is a relational activity. As has already been pointed out (e.g. by [20]), important aspects of a knowledge test belong both to the test itself, and also to the context it is used in. A Deweyan account may relate this to a false dichotomy of 'objective' versus 'subjective'. If we instead embrace an intersubjective view of knowledge, we can focus our analysis on the communicative practices involved in creating and using the knowledge test.

From our described process, we may also conclude that such communicative practices benefit from careful considerations of consensus and disagreements. Analysis of agreement of the consensus estimate and consistency estimates helped create confidence that an unstructured process might not. We can also point to possible dangers of an overly-specific focus for knowledge judgments. Our process benefited from complementing the scoring rubric with holistic assessments, see Section III.

VI. Future work

Both the area of hands-on, as well as the context-specific process, seem to be fruitful topics of future research, particularly combined with a pragmatic action and communication theory. In the larger research project this study is part of, we gather different kinds of student data relevant for understanding the role of hands-on in novice programming students' learning. With a validated knowledge test we hope we will be able to correlate these data with the results from the knowledge test and thus get a deeper understanding of how hands-on may help students' learning to program.

VII. ACKNOWLEDGEMENTS

This project is supported by The Swedish Research Council, grant 2015-01920.

REFERENCES

[1] K. von Hausswolff, "Hands-on in computer programming education," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: ACM, 2017, pp. 279–280. [Online]. Available: http://doi.acm.org/10.1145/3105726.3105735

[2] N. Salleh, E. Mendes, and J. Grundy, "Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 509–525, 2011.

[3] A. E. Tew, *Assessing fundamental introductory computing concept knowledge in a language independent manner*. Georgia Institute of Technology, 2010.

[4] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman, "Identifying student misconceptions of programming," in *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 2010, pp. 107–111.

[5] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, 2013.

[6] E. Soloway, *Studying the novice programmer*. Psychology Press, 2013.

[7] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles, "Identifying important and difficult concepts in introductory computing courses using a delphi process," *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 256–260, 2008.

[8] K. M. Rich, C. Strickland, T. A. Binkowski, C. Moran, and D. Franklin, "K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: ACM, 2017, pp. 182–190. [Online]. Available: http://doi.acm.org/10.1145/3105726.3106166

[9] A. Luxton-Reilly, B. A. Becker, Y. Cao, R. McDermott, C. Mirolo, A. Mühling, A. Petersen, K. Sanders, Simon, and J. Whalley, "Developing assessments to determine mastery of programming fundamentals," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: ACM, 2017, pp. 388–388. [Online]. Available: http://doi.acm.org/10.1145/3059009.3081327

[10] A. Robins, "Learning edge momentum: a new account of outcomes in cs1," *Computer Science Education*, vol. 20, no. 1, pp. 37–71, 2010. [Online]. Available: https://doi.org/10.1080/08993401003612167

[11] A. Petersen, M. Craig, and D. Zingaro, "Reviewing cs1 exam question content," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 631–636. [Online]. Available: http://doi.acm.org/10.1145/1953163.1953340

[12] R. Lister, "Computing education research: Geek genes and bimodal grades," *ACM Inroads*, vol. 1, no. 3, pp. 16–17, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/1835428.1835434

[13] M. C. Parker, M. Guzdial, and S. Engleman, "Replication, validation, and use of a language independent cs1 knowledge assessment," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16. New York, NY, USA: ACM, 2016, pp. 93–101. [Online]. Available: http://doi.acm.org/10.1145/2960310.2960316

[14] S. Shuhidan, M. Hamilton, and D. D'Souza, "A taxonomic study of novice programming summative assessment," in *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95*. Australian Computer Society, Inc., 2009, pp. 147–156.

[15] N. Kasto, "Learning to program: The development of knowledge in novice programmers," Ph.D. dissertation, Auckland University of Technology, 2016.

[16] J. B. Biggs and K. F. Collis, *Evaluation the quality of learning: the SOLO taxonomy (structure of the observed learning outcome)*. Academic Press, 1982.

[17] A. Yadav, D. Burkhart, E. Moix, Daniel Snow, P. Bandaru, and L. Clayborn, "Sowing the seeds of assessment literacy in secondary computer science education: A landscape study," Computer Science Teachers Association (CSTA), Tech. Rep., 2015.

[18] C. Taylor, D. Zingaro, L. Porter, K. Webb, C. Lee, and M. Clancy, "Computer science concept inventories: past and future," *Computer Science Education*, vol. 24, no. 4, pp. 253–276, 2014. [Online]. Available: https://doi.org/10.1080/08993408.2014.970779

[19] L. J. Cronbach, "Coefficient alpha and the internal structure of tests," *Psychometrika*, vol. 16, no. 3, pp. 297–334, Sep 1951. [Online]. Available: https://doi.org/10.1007/BF02310555

[20] S. E. Stemler, "A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability," *Practical Assessment, Research & Evaluation*, vol. 9, no. 4, pp. 1–19, 2004.

[21] J. Dewey, "My pedagogic creed," *School Journal*, vol. 54, no. 3, pp. 77–80, 1897.

[22] ——, *Experience and Nature*. McCutchen Pr, 1925.

[23] ——, "Human nature and conduct: An introduction to social psychology," *Journal of Philosophy*, vol. 19, no. 17, pp. 469–475, 1922.

[24] G. Biesta and N. C. Burbules, *Pragmatism and Educational Research*, 2003.

[25] A. Eckerdal, R. McCartney, J. E. Moström, K. Sanders, L. Thomas, and C. Zander, "From limen to lumen: computing students in liminal spaces," in *Proceedings of the third international workshop on Computing education research*. ACM, 2007, pp. 123–132.

[26] L. J. Höök and A. Eckerdal, "On the bimodality in an introductory programming course: An analysis of student performance factors," in *2015 International Conference on Learning and Teaching in Computing and Engineering*, 2015, pp. 79–86.

[27] P. Gross and K. Powers, "Evaluating assessments of novice programming environments," in *Proceedings of the First International Workshop on Computing Education Research*, ser. ICER '05. New York, NY, USA: ACM, 2005, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/1089786.1089796

[28] J. Sheard, Simon, A. Carbone, D. Chinn, M.-J. Laakso, T. Clear, M. de Raadt, D. D'Souza, J. Harland, R. Lister, A. Philpott, and G. Warburton, "Exploring programming assessment instruments: A classification scheme for examination questions," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11. New York, NY, USA: ACM, 2011, pp. 33–38. [Online]. Available: http://doi.acm.org/10.1145/2016911.2016920

[29] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," in *Acm Sigcse Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 14–18.

[30] A. Eckerdal, "Relating theory and practice in laboratory work: a variation theoretical study," *Studies in Higher Education*, vol. 40, no. 5, pp. 867–880, 2015. [Online]. Available: https://doi.org/10.1080/03075079.2013.857652

[31] B. Shneiderman, "Teaching programming: A spiral approach to syntax and semantics," *Computers & Education*, vol. 1, no. 4, pp. 193 – 197, 1977. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0360131577900082

[32] F. Marton, A. B. Tsui, P. P. Chik, P. Y. Ko, and M. L. Lo, *Classroom discourse and the space of learning*. Routledge, 2004.

[33] A. Luxton-Reilly and A. Petersen, "The compound nature of novice programming assessments," in *Proceedings of the Nineteenth Australasian Computing Education Conference*, ser. ACE '17. New York, NY, USA: ACM, 2017, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/3013499.3013500

[34] "Drjava." [Online]. Available: http://www.drjava.org/

[35] E. O. JR. and H. AGUINIS, "The best and the rest: Revisiting the norm of normality of individual performance," *Personnel Psychology*, vol. 65, no. 1, pp. 79–119. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1744-6570.2011.01239.x