This is the author's version of a work that was submitted/accepted for publication in the following source:


This file was downloaded from: http://eprints.qut.edu.au/57674/

© Copyright 2012 ACM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ICER’12, September 9–11, 2012, Auckland, New Zealand. Copyright 2012 ACM 978-1-4503-1604-0/12/09 ...$10.00.

Notice: Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source:
On the Reliability of Classifying Programming Tasks Using a Neo-Piagetian Theory of Cognitive Development

ABSTRACT

Recent research has proposed Neo-Piagetian theory as a useful way to describe the cognitive development of novice programmers. Neo-Piagetian theory may also be a useful way to classify materials used in learning and assessment. If Neo-Piagetian coding of learning resources is to be useful, it is important that practitioners can easily learn it and then are able to use it effectively. We describe the design of an interactive web-based tutorial for Neo-Piagetian categorization of assessment tasks. We report a study on its effectiveness. Twenty computer science educators completed the tutorial. The average classification accuracy measures on each of the three Neo-Piagetian stages was 85%, 71% and 78%. Participants also rated their agreement with the provided expert classifications, and indicated high agreement (91%, 83% and 91% across the three levels). Self-rated confidence in applying Neo-Piagetian theory to classifying programming questions before and after the tutorial were 29% and 75% respectively. Participants also rated their agreement with the provided expert classifications, and indicated high agreement (91%, 83% and 91% across the three levels). Self-rated confidence in applying Neo-Piagetian theory to classifying programming questions before and after the tutorial were 29% and 75% respectively. The average classification accuracy measures on each of the three Neo-Piagetian stages was 85%, 71% and 78%. Participants also rated their agreement with the provided expert classifications, and indicated high agreement (91%, 83% and 91% across the three levels). Self-rated confidence in applying Neo-Piagetian theory to classifying programming questions before and after the tutorial were 29% and 75% respectively. Participants also rated their agreement with the provided expert classifications, and indicated high agreement (91%, 83% and 91% across the three levels). Self-rated confidence in applying Neo-Piagetian theory to classifying programming questions before and after the tutorial were 29% and 75% respectively.

Categories and Subject Descriptors

K.3 [Computers & Education: Computer and Information Science Education]: Computer Science Education

General Terms

Human Factors, Design, Measurement

Keywords

Programming, CS1/2, Neo-Piagetian, maturity, competence, learning progression, assessment, pedagogy

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Kramer [7] asserted that the key difference between top-performing and under-performing software engineering students is “The ability to perform abstract thinking and to exhibit abstraction skills”. He posed the question “Is it possible to teach abstract thinking and abstraction skills?” If Kramer’s assertion is correct, and if those skills are learnable, then computing educators need methods that (as Kramer expressed it) “measure students abstraction abilities” using tests that “examine different forms of abstraction, different levels of abstraction and different purposes for those abstractions”.

A promising approach to classifying novice programmers’ cognitive development comes from Neo-Piagetian theory [9]. However, that theory has not been evaluated to determine whether it can be used by computing educators to classify reliably learning and assessment materials used in teaching programming. If it is to be useful, the Neo-Piagetian theory should be learnable and practitioners should be able to apply it effectively to classify materials. In this paper, we describe a tutorial system for Neo-Piagetian theory and report our evaluation of how well it enabled computer science educators to learn to use the theory to classify assessment tasks.

After reviewing, in a programming context, Neo-Piagetian theory in the next section of the paper, we then describe our design of an online interactive tutorial system we have developed for Neo-Piagetian theory. We go on to evaluate the reliability of Neo-Piagetian classification of examination questions, by twenty computer science educators who completed the online tutorial.

2. NEO-PIAGETIAN THEORY

The Neo-Piagetian theory of cognitive development [10] is a derivative of Classical Piagetian theory [11]. Classical Piagetian theory focuses on the intellectual development of children as they mature. That is, Classical Piagetian theory focuses on a child’s generic abstract reasoning development across all domains as they grow older. Neo-Piagetian theory instead states that “people, regardless of their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain” [9]. That is, in Neo-Piagetian theory, a person irrespective of age, can display expert reasoning abilities in one domain, but novice reasoning in a different unrelated domain. This is the key difference between Classical Piagetian and Neo-Piagetian theory.

In the context of Computer Science education, Neo-Piagetian theory defines three main stages of cognitive development,
which are (from least mature to most mature) Pre-Operational, Concrete-Operational and Formal-Operational Reasoning.

Those three stages are described respectively in each of the next three subsections. Each subsection contains a description of the Neo-Piagetian stage; an example programming exam question representative of that stage; and an explanation as to why the example question requires a minimum abstraction ability at that Neo-Piagetian stage. The descriptions in these three subsections are adapted in large from Lister [9]. The example questions and explanations were produced collaboratively by the authors of this paper. The descriptions, examples and explanations were incorporated verbatim in the tutorial system, as described later in the paper. (Note that the use of bold font in these next three subsections reflects the use of bold font in the actual online tutorial.)

2.1 Pre-Operational

Typically, pre-operational students can trace code. That is, they can manually execute a piece of code and determine the values in the variables when the execution is finished. However, they tend not to abstract from the code to see a meaningful computation performed by that code.

For the novice who is thinking pre-operationally, the lines in a piece of code are only weakly related. The thinking of the pre-operational student tends to focus on only one abstract property at any given moment in time, and when more than one abstract thought occurs over time those abstractions are not coordinated, and may be contradictory.

A pre-operational student uses inductive reasoning to derive the function of a piece of code by examining input/output behavior. That is, the pre-operational student chooses a set of initial values, manually executes the code, and then inspects the final values.

Example Exam Question: What is the output of the following code?

```java
int a = 3;
int b = 7;
int c = 0;
int[] data = {1, 6, 5, 2, 3};
for (int i = 0; i < data.length; i++) {
    if (data[i] > a && data[i] < b) {
        c++;
    }
}
System.out.println(c);
```

Explanation: This is a tracing exercise where the correct answer can be obtained by a pre-operational student by manually executing the code one line at a time. A higher understanding of the code as a whole is not essential (i.e. realizing that the code returns the number of elements in the data array between the values of a and b). If the array ‘data’ was much larger, a pre-operational student would not be able to manually execute the code to derive the answer.

2.2 Concrete Operational

Concrete thinking involves routine reasoning about programming abstractions. However, a defining characteristic of concrete reasoning is that the abstract thinking is restricted to familiar, real situations, not hypothetical situations (hence the name ‘concrete’).

A concrete operational student can write small programs from well defined specifications but struggles to write large programs from partial specifications. When faced with the latter type of task, the concrete operational student tends to reduce the level of abstraction by dealing with specific examples instead of with a whole set defined in general terms. That is, rather than solving the problem for the general case, they write code to solve a simple subset.

Concrete operational students are capable of deductive reasoning. That is, given a piece of code, a concrete operational student may derive its function just by reading the code. While they may also try manual execution of the code to help confirm this interpretation, they would not simply report the code’s function in terms of input and matching output sets.

Example Exam Question: The following piece of code shifts all elements in the data array one place to the right. The last element in the array is rotated to the front of the array. Modify the function to do the opposite, that is, shift every element one place to the left, and rotate the first element to the last position.

```java
public int[] shiftRight(int[] data) {
    int temp = data[length-1];
    for (int i = data.length-1; i>0; i--) {
        data[i] = data[i-1];
    }
    data[0] = temp;
    return data;
}
```

Explanation: To answer this correctly, a student must understand all the relationships in the given code. A student operating at the concrete level should realize the changes required in storing the correct temporary value, reversing the array direction, changing the iteration bounds accordingly without overshooting, and restoring the correct value in the last position. A pre-operational student may make one or two correct changes, but is unlikely to come up with a complete working solution as he does not understand the operation of the code as a whole, and does not understand all of the relationships between the different abstractions.

2.3 Formal Operational

A person thinking formally can reason logically, consistently and systematically. Formal operational reasoning also requires a reflective capacity - the ability to think about one's own thinking.

Formal operational thinking can involve reasoning about hypothetical situations, or at least reasoning about situations that have never been directly experienced by the thinker. It also involves an awareness of what is known for certain, and what is known with some probability of being true, which in turn allows someone who is thinking formally to perform hypothetico deductive reasoning, that is, the making of a tentative inference from incomplete data, then actively, systematically seeking further data to confirm or deny the tentative inference.

Writing programs is frequently referred to as an exercise in problem solving. Problem solving can be defined as a five step process: (1) abstract the problem from its description, (2) generate subproblems, (3) transform subproblems into
subsolutions, (4) recompose, and (5) evaluate and iterate. Such problem solving is formal operational.

Example Exam Question: Write a program that will read in an arithmetic expression from the console and print out the result. For example, given the input \(3^8/4+(6-(4/2+1))\), your program should output the answer 9 on a new line. The program should gracefully handle all exceptions.

Explanation: To answer this correctly, a student is required to use problem solving skills as described above. This involves logical, consistent, systematic reasoning about programming abstractions in an unfamiliar context to piece together a working solution. A student at the formal operational level would abstract functionality into objects and methods and piece together a solution that is correct and adheres to best-practice design patterns.

2.4 Discussion of the Neo-Piagetian Stages

Piagetian theory also describes a sensori-motor stage, which comes before pre-operational reasoning. Sensori-motor in the context of computer science is defined by Lister et al. as “students who trace code with less than 50% accuracy” [9]. This stage is not considered further in this paper.

Some behaviours of novice programmers are explained by the Neo-Piagetian framework. Ginat [4] observed that when novices were made aware of a bug in their code, by being given a single test case, some novices would patch the code for that test case, rather than fix the general problem illustrated by that test case. Such behavior is to be expected in novices who are reasoning at the concrete operational stage. Hazzan [5] describes a number of situations, in both computing and mathematics, where novices unconsciously reduce the level of abstraction of a concept to make personal sense of the concept. Again, such behavior is to be expected in novices who are reasoning at the concrete operational stage. Kolikant and Mussai [6] described how, when given specific buggy programs on which to comment, some novice programmers viewed the programs as being partially correct. Depending on the severity of the bugs in such programs, the notion of partial correctness is compatible with novices who are reasoning at the pre-operational and concrete operational stages. Kurtz [8] reported that “the levels of late concrete and late formal are strong predictors of poor and outstanding performance, respectively” in an introductory computer science course.

A participant using the tutorial works their way through several phases, as shown in Figure 1. We now describe each of these, explaining the design rationale for each.

3.1 Pre-Survey

The Neo-Piagetian tutorial commences with a pre-survey phase. This asks participants for their level of experience in computer science education (i.e. tutors/teaching-assistants vs. lecturers/professor), and for participants to self-rate their confidence, based on their prior knowledge, at correctly classifying programming exam questions using Neo-Piagetian theory. We refer to this confidence judgment as the Initial Confidence (IC) score, expressed as a percentage (100% indicating complete confidence). Our design incorporates this for three reasons. The simplest of these is that it captures information that can be used to measure participant perceptions of the effectiveness of the overall tutorial. This makes it important for our experimental system. For the long term use of the tutorial, the confidence measures are also important for making effective interpretations of a user's data within the system as we will describe below. The third reason is that it calls upon the user to perform a metacognitive judgement of the feeling of knowing (FOK) and such activation of metacognitive processes can improve learning.

3.2 Initial Overview

In this phase, as shown in Figure 2, participants read descriptions of each of the three Neo-Piagetian reasoning stages, where each description is accompanied by an example exam question representative of that stage and an explanation as to why a typical student would need to be reasoning at that minimum level of Neo-Piagetian reasoning to answer the question correctly. These descriptions and examples were described earlier in Sections 2.1, 2.2 and 2.3. The design of this part of the tutorial system aimed for short descriptions of each level so that the whole description and example fits easily on a typical browser screen and can be read in three to five minutes. This ensures that the system provides a complete overview of the three Neo-Piagetian levels in 10 to 15 minutes.
pants self-rate their confidence at being able to categorize other exam questions requiring a minimum of that level of Neo-Piagetian reasoning. We call this the Prediction Confidence (PC) score. These scores are expressed as a percentage, one for each Neo-Piagetian category. Participants move from one stage to the next to read and rate their confidence on each of the three Neo-Piagetian levels of reasoning before proceeding to the interactive examples.

### 3.3 Interactive Examples

Participants then step through fifteen interactive examples, as shown in Figure 3. In each example, the participant is presented with a programming exam question, and asked to identify the minimum Neo-Piagetian stage of reasoning which would be required by most students to score full marks for that question. This involves using the system in the manner similar to the actual use of the Neo-Piagetian classification for thinking about the level of assessment tasks used in actual teaching.

A screenshot of the interface is shown in Figure 3 where a participant classifies Example 4 as Pre-Operational. The participant self-rates 90% confident (On-Task Confidence (OTC)) and provides an explanation for the classification decision before submitting the form. The tutorial asks for justifications and uncertainties in accordance with the work of Chi et al. showing “Eliciting self-explanations improves understanding”. These explanations are valuable for our research as they enable us to gain insights into people’s reasoning about their classifications and certainty.

After a participant has classified and self-rated an exam question, the interface then presents the nominated classification for the example, along with a justification. The nominated classifications for each of the fifteen examples were developed collaboratively by three of the authors of this paper, all of whom are computer science education researchers with an active interest in cognitive development and learning progression in programming fundamental subjects. That does not necessarily guarantee that each nominated classification is the only correct interpretation of Neo-Piagetian theory, but rather the nominated classifications serve as a reference point from which to base further thought and discussion. However, for the purposes of carrying out the evaluation in this paper, we will regard these as ‘correct’ classifications.

After being shown the nominated classification for a particular example, the participant is then presented with a closed-option response to register their agreement or disagreement with the classification and explanation. This is shown in Figure 5 where the participant is informed that his classification matches the nominated classification (Pre-Operational) for Example 4. We refer to this as the On-Task Accuracy (OTA).

The participant is also shown an explanation of the nominated classification, and has the option to either agree or disagree with these. We refer to this as the Agreement scores (AGR). If the participant disagrees, a textbox is revealed allowing the participant to comment as to why. After submitting this feedback, the participant moves down to the next interactive example, until all fifteen are completed.

### 3.4 Post-Survey

After completing all fifteen examples, participants complete the short Post-Survey. Participants are asked to self-rate their Final Confidence (FC) at being able to classify programming questions according to Neo-Piagetian theory. Participants are also asked to comment on whether they found the tutorial useful and efficient, and whether they would consider using Neo-Piagetian theory when devising future assessment tasks using a series of closed-response questions.

### 4. EVALUATION

We designed an evaluation study to assess two key goals...
4.1 Participants

Twenty participants completed the interactive tutorial. Eleven of these were Computer Science professors or lecturers who have taught or are currently teaching first year computer science subjects. The other nine participants were postgraduate students or computer science researchers who have tutored or are tutoring computer science subjects (also referred to as teaching assistants in some parts of the world).

4.2 Pre-Survey

The average Initial Confidence for all twenty participants was 29%. Out of these twenty participants, only five had encountered Neo-Piagetian theory before in some limited capacity, and they self-rated their existing confidence in understanding and applying the theory between 50% and 74%. The remaining fifteen participants had no prior knowledge of Neo-Piagetian theory and self-rated between 1% (lowest allowed value) and 40% (s.d. 23).

4.3 Initial Overview and Fifteen Examples

The chart in Figure 6 summarises the quantitative results from the initial overview and the completion of the fifteen example questions. The vertical axis is an average percentage score, from 0 to 100, for all twenty evaluation participants. The horizontal axis is grouped into the three Neo-Piagetian categories stages. Within each Neo-Piagetian category, the chart shows four values. These values are, from left-to-right and as described in the previous sections, the participant Prediction Confidence, On-Task Confidence, On-Task Accuracy and Agreement.

Prediction Confidence (PC) average scores were 69%, 63% and 70% for the three Neo-Piagetian categories respectively. These results suggest the descriptions with their embedded initial examples were not sufficient for participants to become confident that they had a solid understanding of the framework. This was further reflected in the post-survey. One participant commented “doubling the number of examples will help with the initial comprehension of the classifications”. Another noted “what might be better is a more comprehensive list of [initial] examples to start off with”. However, the short descriptions and limited initial examples were by design as discussed earlier. The goal of the tutorial is to quickly give an overview of the Neo-Piagetian theory, and have participants develop their understanding through practiced examples and reflection.

On-Task Confidence averages were 82%, 69% and 78%. These scores are significantly higher than the respective Prediction Confidence values, suggesting participants were more confident at classifying each of the fifteen example questions than originally predicted. That is, the process of going through the interactive examples increased self-rated confidence in understanding. The concrete operational category had the lowest average confidence rating, suggesting it may be the most problematic of the three.

On-Task Accuracy average scores were 85%, 71% and 78% respectively. These are close to the On-Task Confidence ratings, suggesting participants were fairly accurate in their self-reflection.

Finally, the Agreement (with the nominated classification) scores of 91%, 83% and 91% respectively, indicate that participants generally agreed with the nominated Neo-Piagetian classifications and explanations after being shown the expected answer in cases where they made a different classification choice. The most debated examples were those targeting the concrete-operational stage.

The chart in Figure 7 shows the average On-Task Confidence, On-Task Accuracy and Agreement scores (as percentages from 0 to 100 along the y-axis) for each of the fifteen interactive examples (numbered 1 to 15 on the x-axis). Ex-
Interactive Example 8

Explain in plain English, using a single sentence, the purpose of the following function.

```javascript
function whatDoIDoFunction(x, array) {
    var y = 0;
    var i = 0;
    for (i = 0; i < array.length; i++) {
        if (array[i] == x) {
            y++;
        }
    }
    return y;
}
```

This example had a nominated classification of Pre-Operational, which was explained as follows:

While this is not a tracing exercise, a student might solve this by substituting some specific values for `x` and the array and tracing one or two iterations of the loop, which should reveal the purpose of the code (i.e., inductive reasoning). A concrete-operational student will answer this without the manual tracing (i.e., deductive reasoning), although this would be hard to distinguish from the answer. This code is a particularly simple iterative process on an array. Based upon cues such as the use of “for”, a pre-operational student might surmise that the code scans across the array, without completely understanding exactly how it scans across the array. After making such an assumption, a student can then answer the question by focusing solely upon the “if” within the loop, and its associated increment to variable “y”. The student need not be worried about how successive iterations of the loop will affect each other. Such a student might be considered to be late pre-operational.

Thirteen participants initially classified this as concrete operational, and one as formal operational. Participants who selected concrete operational commonly stated that the student is required to have a deeper understanding of the code to answer correctly. Comments included: “requires holistic understanding of a simple, specific piece of code”; “the student needs to be able to reason about the high level operation of this code”; “the student needs to understand the relationships between the lines of code and how the code works as a whole”; and others along similar lines. This shows participants may have been lead to believe that any code exercise that is not tracing requires higher levels of reasoning than pre-operational. However, after being presented with the nominated classification and nominated explanation, 80% of participants accepted the explanation. The four participants who disagreed with the nominated classification noted that it is a border-line case which is difficult to clearly distinguish. “I believe this is another borderline question (that is, bordering between pre-operational and concrete operational). In the explanation of what these two categories are, there is the idea of using specific input or concrete examples to make inductive leaps. It is unclear at what level of simplicity/complexity we start to cross into the concrete operational category. Perhaps this is a community-defined border (and it does not need to be an absolute border).”

Example 2 and Example 9 (which are available online) had the second and third lowest OTA scores with 11 and 12 participants answering correctly, respectively. Example 2 was nominated as a pre-operational task, and Example 9 as concrete-operational. In both cases, the majority of participants whose answers did not match the nominated classification, instead opted for concrete-operational and pre-operational respectively. This suggests two possibilities. One approach is for the tutorial to differentiate more clearly between the pre-operational and concrete-operational stages. An alternate approach is to acknowledge that a three-level system, with solid theoretical psychological foundations has considerable merit, particularly that of simplicity; however, it does not provide the nuanced distinctions that our participants saw between this question and others with clearer categorizations. This may be partly captured in the level of disagreement (AGR) for the more contentious questions. However, even this part of the interface did not enable participants to indicate that they could see the merit in the reasoning provided but did not feel that the classification for this example was clear-cut. Indeed the explanation above makes it clear that the expert explanation rated the class of Q8 as high in the Pre-Operational level. We could refine the interface on the confidence measures to allow participants to distinguish between lack of confidence in a classification due to their self-perceived lack of knowledge and their view that the classification itself was not clear-cut. Of course, we would need to assess the benefits of this against the additional complexity of the interface.

Figure 7: Aggregate participant results for each of the fifteen interactive examples

4.4 Post-Survey

The average Final Confidence was 75% with a standard deviation of 15. Nineteen of the ten participants self-rated their understanding between 50% and 90%, but one participant self-rated at 29%. The 75% Final Confidence average is a large improvement on the 28% Initial Confidence result from the pre-survey. This, together with the On-Task Accuracy scores, indicates the tutorial was highly effective at introducing the Neo-Piagetian theory, and quickly building
a good level of applied understanding. The tutorial however did not lead to perfect confidence or understanding, suggesting either more training is necessary or that the neo-Piagetian framework is still a work in progress.

In the closing feedback comments, all participants were generally pleased with, and positive about, the effectiveness of the tutorial, although some suggested more examples are needed to solidify understanding. The post-survey also presented participants with a set of six yes/no check-box statements that they could either agree with or disagree with. The results for the 20 participants for these check-box statements are coded below with the overall number agreeing, followed by the breakdown shown as (lecturer/professors, tutors) from the total of 11 and 9 respectively:

- 12 agreed (7,5) that “the tutorial helped me change the way I think about programming assessment” [A]
- 14 agreed (7,7) that “I now have a better appreciation of the different competence levels required to solve tasks” [B]
- 12 agreed (7,5) that “I may consider using Neo-Piagetian theory for classifying some of my own exams or assessments in the future” [C]
- 4 agreed (3,1) that “There is too much ambiguity to use Neo-Piagetian theory for classifying programming tasks” [D]
- 17 agreed (11,6) that “I found this exercise useful” [E]
- none agreed that “I found this exercise to be a waste of time” [F]

Overall these comments suggest participants found the tutorial useful and that they may consider using some of the lessons learned about cognitive reasoning, abstraction, and Neo-Piagetian classification in some of their own computer science assessment exercises. This is particularly so for the lecturer/professor participants - all eleven of whom indicated that they found the exercise useful and many of whom also agreed to consider using the framework for classifying their own future assessment tasks “It was useful, especially with the detailed explanations for each example. Importantly I think I could do a better job of formulating exam questions at the level intended”.

4.5 Individual Participant Results

Figure 8 shows the result averages for each participant. The columns from left to right are: Participant ID (PID); Experience (EXP) where L/P is Lecturer/Professor and T is tutor or teaching assistant; Initial Confidence (IC); Prediction Confidence for the three Neo-Piagetian levels (PC1 for pre-operational, PC2 for concrete operational, and PC3 for formal reasoning); On-Task Accuracy (OTA); On-Task Confidence (OTC); Agreement (AGR) as an average across the fifteen questions; the Final Confidence (FC); and the post-survey feedback statements discussed above (POST).

The results show the eleven Lecturer/Professor participants had a higher Initial Confidence in Neo-Piagetian theory than the nine tutors. The Lecturer/Professors also had higher average results for all other categories. Additionally, PID33 seems to have struggled the most. This tutor had no prior knowledge of Neo-Piagetian theory, and self-rated very conservatively on all confidence metrics (PC, OTC and FC all range between 22 and 37%), even though On-Task Accuracy was 60%.

The final key issue we report is the time taken by participants to complete the tutorial. As noted earlier, some participants (5) completed the tutorial in private in their own time. For these participants we were not able to accurately discern interruptions and distractions from focused attention spent on the tutorial. However, the remaining participants (15) completed the tutorial in one sitting and under supervision, and we were able to capture the average time taken to complete the tutorial as 67 minutes, with the slowest participant taking 96 minutes.

5. CONCLUSIONS

Ensuring learning activities and examination exercises are at an appropriate level of difficulty is critical for effective teaching and effective learning in computer science. Recent observations suggest current practices in computer science education are not optimal in this respect. A systematic framework is required to enable computer science educators to target learning at the correct level of competence to achieve and maintain flow, and to accurately grade student abilities. A number of such frameworks exist in the educational domain, the most widely-cited of which is Bloom’s Taxonomy. Attempts at using these frameworks in computer science education have not been overly successful to date. The
main reasons for this being a lack of understanding of the frameworks by teaching academics, lack of contextualization to computer programming, and inappropriateness of frameworks to measure the important characteristics of cognitive development in the computer science discipline.

The Neo-Piagetian theory of cognitive development, however, is a promising candidate for use in computer science as it deals directly with different stages of abstraction and abstract reasoning. These characteristics have been identified as key differentiators between top-performing and underperforming students. It seems to offer the potential to serve as a way for computer science educators to build a vocabulary to talk about and to think about the level of difficulty and abstraction of learning activities and assessment tasks. However, if this is to have real value and impact, computing educations must be able to quickly learn to use it effectively. It offers particular value for the coding of the level of programming exam questions, but also have value for other learning activities.

As such, we have created an online web-based interactive tutorial to quickly and effectively up-skill computer science educators in Neo-Piagetian theory. The evaluation with twenty participants revealed that the tutorial succeeded in significantly increasing understanding of this framework through a small number of contextualized examples and descriptions. Participants also commented positively on the usefulness and relevance of Neo-Piagetian theory, and suggested they would consider using it to plan the development of future examination exercises. This indicates that Neo-Piagetian theory offers promise in a computer science context, that computer science educators are interested in learning about it and using it in practice, and that our tutorial system provides an effective introduction to this framework in just over one hour of learning time.

Participants however commented that more examples would be desirable to strengthen their understanding further. A revised version of the tutorial may thus have a large database of different examples which are used at random, and participants may complete any number of these until they feel confident in their understanding. Additionally, participants may benefit from being able to submit some of their own example exam questions and have these classified by other users of the tutorial system. This would be useful in generating a database of real programming questions as used in real exams, which are classified using the Neo-Piagetian framework by fellow computer science educators. Such a repository would be a valuable resource in learning the framework, in gaining inspiration for constructing new exam exercises and in developing a commonly accepted and commonly understood systematic approach to measure and specify competence levels in computer science. These features are currently under active development as part of the [...text omitted for anonymous review.]

6. REFERENCES


