The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes through the Lens of Situated Cognition

Quintin Cutts
School of Computing Science
University of Glasgow
Glasgow, Scotland
+44 141 330 5619
Quintin.Cutts@glasgow.ac.uk

Sarah Esper, Marlena Fecho,
Stephen R. Foster, Beth Simon
Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA 92093
{sesper, srfoster, mfecho, bsimon}@cs.ucsd.edu

ABSTRACT
We report on a post-hoc analysis of introductory programming lecture materials. The purpose of this analysis is to identify what knowledge and skills we are asking students to acquire, as situated in the activity, tools, and culture of what programmers do and how they think. The specific materials analyzed are the 133 Peer Instruction questions used in lecture to support cognitive apprenticeship – honoring the situated nature of knowledge. We propose an ***Abstraction Transition Taxonomy*** for classifying the kinds of knowing and practices we engage students in as we seek to apprentice them into the programming world. We find students are asked to answer questions expressed using three levels of abstraction: English, CS Speak, and Code. Moreover, many questions involve asking students to transition between levels of abstraction within the context of a computational problem. Finally, by applying our taxonomy in classifying a range of introductory programming exams, we find that summative assessments (including our own) tend to emphasize a small range of the skills fostered in students during the formative/apprenticeship phase.

Categories and Subject Descriptors
K.3.2 [Computer Science Education]

General Terms
Human Factors

Keywords
taxonomy, CS0, CS1, CS2, situated cognition, cognitive apprenticeship, deliberate practice

1. INTRODUCTION
Situated cognition is a learning theory in which learning is seen not from the isolated cognitive, conceptual, or abstract perspective of the individual learner, but rather as situated within "the activity, context, and culture in which it [learning] is developed and used." [1] This viewpoint would seem to resonate with the computing education community. We are a discipline with a strong professional focus, grounded in the development of a tool (i.e., the computer) for use in solving problems for the benefit of society. As such, it is *implicit* in our curriculum that we seek to apprentice students in becoming (more) masterful computing professionals. Could instruction be improved by developing more *explicit* learning goals generated through the lens of situated cognition?

In this paper we consider the learning materials used in a pilot of the CS Principles course, under development in the United States. It provides an interesting case because of two factors:

1. The course sought to provide general education (not pre-professional) computational thinking skills. As developers of the course, we were vigilant in considering our target audience. Our mantra "if this is the last computing course a student ever takes, what do I want them to know" resulted in an emphasis on acculturating students into the ways computing professionals see, understand, and solve problems – because it is these ways of thinking that will serve them as they engage with computation throughout their lives. The ability to write programs was important primarily in service to the acculturation goal, not as a goal in its own right.
2. The course was highly successful in how it impacted students' perceptions of future computing use. In [2] we report on the ways students said they could solve problems better, transfer what they had learned to new situations, and more generally, how their confidence had increased and that they could now see technology in a new way.

Because of these factors, we have performed an analysis of the course learning materials to provide an explication of what it was that students were engaged in doing that might have contributed to these results. Using situated cognition theory in this analysis is appropriate because the course design centered on Peer Instruction – a pedagogy that supports cognitive apprenticeship in the classroom. The 133 clicker questions we analyzed were developed by the instructor through daily consideration of the programming concepts and constructs presented in the text-book in light of the following questions: "How can I get students to see this concept/construct the way I see it? To understand why I use it the way I do to analyze, solve, or debug problems?" Because clicker questions are presented as multiple-choice questions (MCQs), this required focusing on one small aspect of computing thinking at a time.

The ***Abstraction Transition Taxonomy*** resulting from our analysis forms a description framed by situated cognition: the amalgamation of activities, tools, and culture that the instructor felt would acculturate the students into the world of computing. We express our analysis in terms of a taxonomy because of the

ICER'12, September 9–11, 2012, Auckland, New Zealand.

nature of the analyzed materials – a number of taxonomies exist to classify assessment questions. Taxonomies are useful because they afford communities a shared vocabulary for discussing learning. We propose that the categories of clicker questions define both a more explicit and more extensive set of learning goals, a set that we should adopt for introductory programming classes. Nonetheless, we find, through analysis of summative assessment items from 7 different introductory programming courses, that a much-restricted set of these learning goals are measured by final exams – including our own.

2. MOTIVATION

2.1 Expertise Development

The goal of any introductory-sequence programming course is not simply to create students who can generate working programs. Rather, more holistically, we want students to begin to see problems as programmers do and learn to apply programming concepts and constructs in the "right way" to solve computational problems. Perhaps because we are aware of our responsibility to produce computing *professionals*, compared to disciplines such as biology or math, our curriculum embraces the intertwining of theory and application and we start students immediately down the path of 10,000 hours of deliberate practice required to develop expertise [3]. That is, we start them solving problems by writing computer programs as they learn the vocabulary/language and concepts underpinning the discipline. We posit that our typical educational practices could be better informed by the literature on expertise development.

Let us consider a long-utilized method for developing expertise – the master-apprentice model. Consider an analogy: the apprenticeship of tailors.[1] Apprentice tailors work closely with their masters in two important ways we want to contrast to common models of programmer education:

1. ***Scaffolding.*** Tailors are scaffolded in the development and practice of their skills by the master – starting with tasks like ironing which engage them in legitimate peripheral participation (the opportunity to observe and acculturate while making a contribution) [4]. They are also set smaller, simpler tasks – they are not assigned to make an entire suit, they might be given various piecework a bit at a time; the button holes, cloth selection and preparation, fitting, etc.
2. ***Process.*** Tailors are not judged merely on the final outcome or garment. The master works in close proximity to the apprentice with the responsibility to observe critical professional skills: processes, analysis and decision-making, and intermediate results.

We propose that there is a lesson for us to learn from the master tailor. Computing instructors focus far too much on the final product – the small program/the simple garment. We do not provide enough direction in the preparatory steps – moreover, while we may "teach" students about various steps, we don't provide enough guidance on the specific masterful ways we approach the steps, the ways in which we evaluate our activities in the steps, and the considerations we take in deciding upon steps. We require our students to experience these for themselves through their experiences doing programming problems. We really only look at their finished products –we don't go through the process with them and observe whether they are thinking and deciding as we would like – not just "doing" as we do.

[1] An example inspired by the work of Jean Lave, but expanded and interpreted by the authors.

2.2 Defining Learning Outcomes through Situated Cognition Theory

Situated cognition theory embraces the notion of learning and knowing as inseparable from the way in which that knowledge will be applied in real life. This stands in contrast to the individualistic view of learning, where learners can be taught conceptual knowledge abstracted from the situation in which it is learned and used [1]. Instead, situated cognition theory states that *knowledge is situated in a triad of activity, tools, and culture*: that it's not just what steps to take using what resources; a critical aspect comes from the context and culture in which that knowledge is to be exercised.

In the language of situated cognition theory, in computing education we engage students in *activities* using computing *tools*. What is missing is the *cultural* portion – the fact that our students are not doing real piecework that will contribute to a real suit for a real customer and, most importantly, under the auspices and detailed guidance of a master. To support development of expertise we need to consider all three components – activities, tools, and culture.

Figure 1 proposes some programming-specific interpretations of activity, tools, and culture. Culture is critically interdependent in this creation; it determines how and why practitioners choose the tools they do, what activities they engage in and how they know to make those choices. As stated in [1], "[t]he activities of many communities are unfathomable, unless they are viewed from within the culture." Perhaps this sheds light on bizarre programming behaviors, completely unexpected questions, and often intense frustration expressed by so many of our students.

Activity	Running a program, observing program behavior or output, inspecting code, making edits, compiling, hypothesizing
Tools	Programming constructs, programming concepts, IDEs
Culture	Knowing to match up runtime behavior with static codebase, knowing what variables to trace, knowing the right way to decompose a problem, knowing to look at a problem in light of what constructs are available to solve it.

Figure 1. The core components of situated cognition, using programming as an example.

To shed light on the difference between activity and culture as defined in Figure 1, consider the difference between novice and expert debugging. Instructors joke about stories of novices using the following activities: run program, observe output, note that it's not what was expected, make an edit at random, recompile, run again, and so on. This is a novice culture. The expert would: run the program, observe the output and note that it wasn't what was expected, examine the code and the problem statement or maybe add diagnostic print statements in order to develop a hypothesis as to the problem, run the program with particular test data to check the hypothesis, and so on. The individual activities are simple and should be relatively easy to enact. It is the choice of activities and the way the activities are combined that defines the difference between novice and expert, and hence the cultural aspect.

One of the primary manners of supporting learning within the lens of situated cognition is through the development and support of communities of practice [5]. A community of practice is made up of experts or practitioners who share a profession. Member

development (from novice to more expert) is embedded within all of the shared information and activities of the community. Members' expertise is developed through interactions with others in the community, while working on real problems of community interest, expressed using the community's own vocabulary..

Communities of practice have inherent conflicts with traditional modes of academic instruction. Most critically, the members of the community (the students) are novices (or relative novices in the course in question). The expert is the lone instructor (who may or may not be involved in a professional community of practice in the subject area). Additionally, the customs of individual assessment often lie in conflict with authentic practice processes and problems.

Cognitive apprenticeship is a method developed by [4] seeking to "try to enculturate students into authentic practices through activity and social interaction in a way similar to that evident – and evidently successful – in craft apprenticeship" [1]. In this course, we employed Peer Instruction to support cognitive apprenticeship [6]. We deliberately and consciously developed clicker questions *not* with the goal of asking students to program, but asking them to perform various tasks and consider various analyses that the instructor felt would help them see how the computing community sees such things.

In this paper we discover how incorporating the consideration of culture helped generate a more specific and larger set of desired abilities and knowledge than are traditionally discussed in literature of desired competencies of introductory programming students (see related work). Since these competencies have mostly been studied within the realm of summative assessment (exam) classification and evaluation of student performance, this is not too surprising. However, we believe the community will benefit from consideration of this new approach and from comparison of these competencies with those previous discussed.

3. RELATED WORK

Learning Taxonomies. Learning taxonomies are important for computing education because they give the community a vocabulary to use when discussing student understanding and learning – and curriculum supporting it. Taxonomies should support educators in having conversations and in reporting on their courses and efforts to improve what goes on in them. Bloom's Taxonomy is a highly popular taxonomy. It consists of 6 levels to describe students' cognitive development. In 2001 Anderson produced a revision of Bloom's Taxonomy consisting of 24 categories within two dimensions. The Knowledge Dimension consists of Factual, Conceptual, Procedural and Metacognitive knowledge and the Cognitive Process Dimension consists of Remember, Understand, Apply, Analyze, Evaluate and Create. A more detailed description of each category can be found in [7] (pp 29-31).

The Revised Bloom's Taxonomy's value has been called into question in computer science [8] and has been modified to include "higher application" in order to make it relevant to the field [9]. Other taxonomies, such as SOLO [10] assess cognitive level through student response-type. Our work differs from these in that we move our focus beyond assessment of students' cognitive skill development to consideration of their development in the situated goal of coming to think like and perform the actions of a programmer. Most recently, work by Lister, et. al, has moved to consider desired programming skill development through Piagetian constructivist learning theory [11].

Apprenticeship and Deliberate Practice. Our clicker questions sought to support students during the apprenticeship phase of learning. Similarly, Faulkner created a more authentic experience for novices through *Worked Examples*: "These are designed to encompass the spectrum of the problem solving process:

- observing the application of programming concepts
- observing authentic problem solving
- cooperative problem solving" [12]

Furthermore, Faulkner asserts that the instructor must display the problem solving in the same environment that the students will be working. [12]

Bareiss takes an approach similar to *Worked Examples* where students are engaged in cognitive apprenticeship learning techniques through coaching. [13] More specifically, Bareiss defines five techniques needed to coach students: Modeling, Scaffolding and Fading, Articulation, Reflection and Exploration. Scaffolding and Fading is similar to *Worked Examples* in that it focuses on Task Design, Direct Guidance and Feedback.

Worked Examples and coaching are two ways to encourage deliberate practice [3] in novice programmers. As described by Ericsson:

> "The theoretical framework of deliberate practice asserts that improvement in performance of aspiring experts does not happen automatically or casually as a function of further experience… The principal challenge to attaining expert level performance is to induce stable specific changes that allow the performance to be incrementally improved." [3]

Deliberate practice involves practicing with the goal of making small improvements at each step, and more importantly, getting feedback along the way to improve the practice when improvement attempts are unsuccessful.

Acculturation. There have been efforts to acculturate students in programming using professional-world approaches. Pair programming [14] is a technique where one student (the driver) works on the lower level parts of programming (e.g., coding) and the other student (the navigator) works at the higher-level (e.g., design and integration). The students switch over being driver and navigator throughout the assignment. Pair programming has been effectively used in the classroom [15] and is an effective way to encourage students to engage in confident practice while learning and consequently doing better in their programming. [16]

Coding Dojos [17] harken back to something like the master-apprentice model to support acculturation. Seemingly not formally studied, Coding Dojos are places where programmers can go and watch others program or be watched programming. The idea is if novices are watching experts they can see the choices experts make and if experts are watching novices they can guide them. Compared to pair programming this emphasizes the community and culture aspects of situated cognition.

4. CREATING THE AT TAXONOMY

4.1 Methodology

4.1.1 The Transition Levels

We developed the ***Abstraction Transition (AT) Taxonomy*** through analysis of learning materials from a CS0-type course teaching Alice serving ~570 students as a general education course at a large research-intensive institution in the US. The learning materials we analyzed were multiple-choice clicker

questions (MCQs) posed *during lecture* for students to answer and analyze in groups (via the Peer Instruction pedagogy [6]). These formative assessment questions sought to support learning through cognitive apprenticeship. However, most exam questions for this course were modeled very closely on these MCQs.

All but one author (Foster) was involved in the teaching and delivery of the course – creating questions and interacting with students discussing them during lecture. After the end of the term, Fecho reviewed the in-class clicker questions and generated categories stemming from her instructional experience with those questions. She identified three levels of abstraction in the questions: English, CS Speak, and Code.

The primary instructional team (Simon and Cutts) did not consciously consider "levels of abstraction" in clicker question development but, in retrospect, this focus was not surprising. On the second day of class, the instructor used Figure 2 as an overview to students of what they would be asked to do in the course.

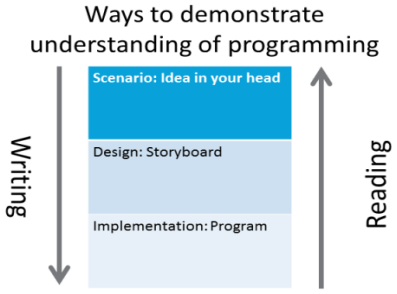

Figure 2. Slide presented in second lecture revealing (in hindsight) instructional focus.

In our analysis we refined Scenario to English, Design to CS Speak and Implementation to Code. We use English instead of specification to indicate that no specialized language is used describing the intended behavior. In Alice, an example of English would be "make a skater spin around 3 times". In CS Speak this might be "have the iceSkater turn 3 revolutions". In code this would be:

In Alice, for the simplest cases, English and CS Speak can be quite similar, and are differentiated by the specific methods and objects defined in Alice.

Through continued discussion we developed a taxonomy that describes those activities and tools, and ways of thinking about *using* those activities and tools, that the instructor felt students need to engage with in order to understand how computing and computing professionals work. A clear finding involved our emphasis on developing students' skills in *transitioning between levels of abstraction*. This was especially evident because of our use of MCQs – with their "stem" and "option" components. Figure 3 represents a transition from the CS Speak abstraction level to the Code abstraction level.

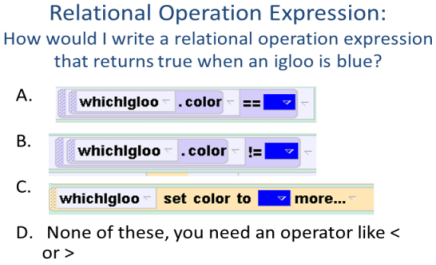

Figure 3. Example MCQ: Transition from CS Speak to Code.

To better clarify the difference between English and CS Speak, we also provide a question transitioning from English to CS Speak. This question is asked in English and the answer choices require the students to use their understanding of CS Speak:

> *Suppose there are customers waiting in line at the store. You want to serve each customer one at a time, so each one should walk to the counter one at a time. How could you do this? A. Use a DoTogether tile, B. Use a DoInOrder tile, C. Use a ForAllTogether tile, or D. Use a ForAllInOrder tile.*

After categorizing a few sample questions, we began to acknowledge that some question and answer sets remained within one level of abstraction, mainly CS Speak or Code. However, these non-transition questions still lie explicitly within an abstraction level and bring out interesting issues from the point of view of cognitive apprenticeship. CS Speak was especially interesting – as it represents the "lingo" of expert programmers. We found we asked two types of tasks with these questions. We term them: CS Speak Apply and CS Speak Define. That is to say, a CS Speak Apply question would engage students in identifying what CS Constructs/Concepts were involved in an algorithm or in deciding which/how CS Constructs/Concepts might solve a goal described by a CS Speak description. For example, CS Speak Apply would be:

> *If we write a method called drive, which would not makes sense as a parameter to control how drive occurs? A. Destination, B. How Fast, C. Which car, or D. Car color*

A CS Speak Define question, on the other hand, would engage a student in explicitly reflecting on what CS Construct/Concept does or what it is used for in the programming community. For example, a CS Speak Define question would be:

> *Which of the following is the best explanation of what makes a good parameter? A. It's something that supports common variation in how the method is done, B. It's got a meaningful name, C. It can be either an Object or a number, or D. It helps manage complexity in large programs*

Questions that remained in the Code abstraction level were code-tracing questions. Although this may not be a transition through abstraction levels, it represents a skill all experts have and sometimes employ in that it requires tracing of code without any analysis or deeper understanding past that of the logic required to solve the problem. For example, a question in this transition would be:

> *How far up will the bee move in the second instruction, given that the tulip is 0.3 meters high and our fly is 0.1 meters tall? A. 0.2 meters, B. 0.3 meters, C. 0.35 meters, D. It's not possible to tell, or E. I don't know*

Furthermore, we agreed that due to the nature of Alice, there might be questions that were unrelated to computing concepts and solely related to Alice-specific issues. These, we gave a classification of "other" and did not expect these types of questions to appear in other programming languages. We do not report on these here.

4.1.2 Different Types of Questions

While performing this analysis we noted an orthogonal question classification descriptor: rationale questions and mechanism or definition questions. We summarized these as *why questions* and

how questions (respectively) and they are defined in Table 4. *How questions* are a norm in computing education. The *why questions* are a natural outgrowth of making explicit the need to rationalize, culturally, decisions of activity and tool use in programming problems. That is, rather than just *hoping* students internalize how programmers think, we use *why questions* to *explicitly* engage students in discussion and evaluation of various programmer rationales.

An example of a *why question*, with AT level 32, shows two code snippets that have the same outcome and then asks:

What is the BEST explanation of why one is better than the other? A. Option 1 is better because it is shorter, B. Option 1 is better because it does the least number of "checks" (or Boolean condition evaluations), C. Option 2 is better because it makes clear exactly what the "checks" (or Boolean condition evaluations) are, or D. Option 2 is better because it has a regular structure with empty "else" portions

On the other hand, Figure 4 is an example of a *how question*, with AT level 13 asking the student to choose code that would match the description.

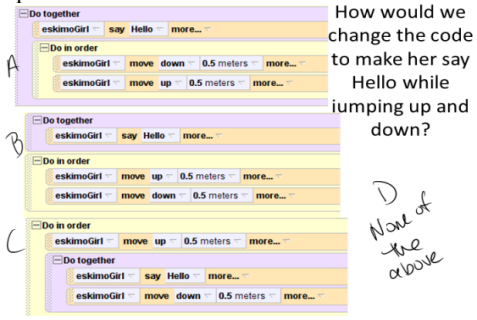

Figure 4. Example question showing a how-type question

To improve the clarity and support reproducibility of AT Taxonomy categorizations, we selected a random sample of 20% (27/133) of all the MCQs asked over the term and had two authors categorize and discuss them based on our taxonomy. From that the two authors iteratively refined the taxonomy. Once finalized, those two authors individually categorized a second random 20% (28/133) of the MCQs for both transition number and type. The authors reached an 87% inter-rater reliability (counting matches for agreement on transition number and agreement on type). Finally, one of the authors coded the remaining 60% of the MCQs from the CS0 course.

4.2 Results

The transition levels used to categorize questions are defined in Table 3. The types (why and how) used to further categorize questions are defined in Table 4. Together, Tables 3 and 4 define the Abstraction Transition Taxonomy. Table 1 below shows the final results of applying the classification scheme to our in-class Peer Instruction MCQs. Due to lack of space, we don't show the breakdown of how and why questions in each category, but overall 21% of questions were *why* and the rest were *how*.

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		6%	9%
CS Speak: 2	2%	8%(A) / 28%(D)	6%
Code: 3	12%	15%	9%

Table 1. CS0 in-class MCQ Distribution. Overall 21% were why questions

To read the chart, look in the row for the level of the question stem, and then move to the column that indicates the option level. This shows that 9% of the questions asked in lecture through MCQs had a question stem in English and the answers were in Code, making 9% of the questions an AT level 13. Similarly, 12% of questions were AT level 31, 6% were 23 and 15% 32.

5. APPLYING THE AT TAXONOMY

5.1 Methodology

Returning to our original goal, we used the AT Taxonomy to help both overview and tease out the manners in which this course sought to help students develop computational thinking practices – or to reach a basic level of understanding of the programming community. We noticed that our CS0 course was fairly distributed amongst all the transitions, with perhaps a surprising focus on use (2A) and understanding (2D) of CS Speak.

With this enlightened view of the course, we explored how our summative assessment (final exam) differed from or matched our formative assessment (in-class MCQs). We were also interested to explore other introductory CS courses' (CS0, CS1 and CS2) summative exams to see a) if our taxonomy would be applicable to others' assessments and b) if our course differed from theirs (as measured by summative assessment).

We analyzed 7 exam sets taken from 4 sources: recent assessment literature, a standardized exam from the Advanced Placement series in the US, a complete exam with 8 questions from a large mid-west US Institution found on the web, and our own CS0 exam. Of note, the CS2-DCER data represents 3 complete exams from the year 2009 (the most recent available) accessed through the DCER project [18]. Table 2 summarizes the datasets.

Dataset Name	Language	Number of Questions	Complete or partial set
CS0 - Our Exam	Alice	37	Complete
CS0 - Meerbaum-Salant	Scratch	5	Partial
CS1 - Lister	Java	12	Partial
CS1 - Lopez	Java	24	Complete
CS1 - APCS	Java	22	Partial
CS1 - R1	Java	8	Complete
CS2 - DCER	Java	143	Complete

Table 2. Summary of Datasets. The CS2 dataset is 3 exams.

5.2 Results

5.2.1 Transition Distribution

As noted in Section 4.1, the in-class clicker questions from our CS0 course are fairly distributed across the taxonomy categories. How were these skills reflected in summative assessments? Tables 5 through 11 show the AT Taxonomy applied to the exams.

CS0. It is thought provoking that our exam question distribution was not reflective of the kinds of questions asked in class. Rather, the exam is almost "all about code" – all but four (related) questions involve an AT level 3. This suggests that even if an educator is fully aware of the need for more diverse cognitive apprenticeship in programming culture and tasks, the norms of examinations may send a very different message to students about what is important.

Transition Level	Description
12	**English-CS Speak:** Given an English description of a scenario or goal, choose a technical description in "CS Speak" of the process to achieve the goal in the form of an algorithm or storyboard.
23	**CS Speak-Code:** Given a technical description (CS Speak) of how to achieve a goal, choose code that will accomplish that goal.
13	**English-Code:** Given an English description of a scenario or goal, choose the code that will accomplish that goal.
32	**Code-CS Speak:** Given code, choose either a description in CS Speak of the goal of the code, or choose which coding constructs are used within the code.
21	**CS Speak-English:** Given a description in CS Speak or a coding construct, choose an English description that describes what the CS Speak does or the goal that it accomplishes.
31	**Code-English:** Given some code, choose an English description that describes the goal of the code, not the step-by-step process, but the overall goal.
3 (Apply)	**Code:** Given code and conditions, choose a result from executing the code. This is "code tracing" and does not imply overall goal, but simply the execution of the code.
2 (Apply)	**CS Speak:** Given a CS Speak description, choose the coding constructs that are present (or vice versa).
2D (Define)	**CS Speak:** Provided coding constructs choose a technical description of their purpose or how they work.

Table 3. Definition of Transition Levels

The numerical coding in the leftmost column represents the particular transition where 1 is English, 2 is CS Speak and 3 is Code. Thus, an AT level 23 would be a transition from 2 (CS Speak) to 3 (Code). AT levels that do not follow this scheme are: CS Speak Apply (AT level 2), CS Speak Define (AT level 2D), and Code (AT level 3).

Type	Description
Why	Choose a **rationale** for why a statement or answer choice is correct or incorrect.
How	Choose an unambiguous answer that depends on **mechanism or definition**.

Table 4. Definition of Types

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		0%	27%
CS Speak: 2	0%	11%(A)/0%(D)	11%
Code: 3	8%	27%	16%

Table 5. CS0 Final Exam

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		0%	0%
CS Speak: 2	0%	0%(A)/20%(D)	40%
Code: 3	20%	0%	20%

Table 6. Meerbaum-Salant

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		0%	0%
CS Speak: 2	0%	0%(A)/0%(D)	42%
Code: 3	0%	0%	58%

Table 7. Lister

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		0%	13%
CS Speak: 2	4%	8%(A)/8%(D)	8%
Code: 3	13%	17%	29%

Table 8. Lopez

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		5%	9%
CS Speak: 2	0%	0%(A)/0%(D)	14%
Code: 3	0%	23%	50%

Table 9. AP CS A

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		0%	13%
CS Speak: 2	0%	25%(A)/0%(D)	50%
Code: 3	0%	0%	13%

Table 10. Midwest R1 Institution

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1		2%	0%
CS Speak: 2	0%	46%(A)/29%(D)	3%
Code: 3	0%	5%	14%

Table 11. DCER

Of the set of assessment questions reported in Meerbaum-Salant's Scratch paper [19] 60% of them fall in the 23 or 3 AT level. The course is therefore assessing the students' coding ability – can students write or trace code, possibly given a storyboard or pseudocode description. The focus of the course is to introduce computing concepts, but it is clear that the assessments are also focused on code.

CS1. Both the Lister [20] and Lopez [21] datasets reflect efforts to study student abilities on commonly desired programming skills across institutions and countries. Lopez was a part of the

BRACElet [22] project where assessment of Lister's questions led to the development of Explain in Plain English (EiPE) questions. The Lopez dataset is the only set that reports assessment of a range of AT levels. This is likely because he sought to explore a new type of question within the framework of existing exam questions. Lopez's identified types of questions are: EiPE (AT31), Code Writing (AT13), Tracing 1 (AT3) and Tracing 2 (AT3).

Both the APCSA sample and the R1 exam show strong emphasis on level 3 activities. 50% of the sample questions for the APCSA exam are AT 3 level requiring code tracing with no abstraction transition and no engagement with contextualization of the problem. In the R1 exam, we also see the emphasis on transitions focusing on the code – 75% of questions fall into AT 13, 23, or 3.

CS2. Quite strikingly, different skills seem to be valued in CS2 exams. These exams represent 3 complete exams from the DCER international dataset. Nonetheless, we find a switch in emphasis from code (and transitions to/from it) to CS Speak; asking students to apply appropriate computing concepts or indicate understanding of the purpose of use of those concepts.

5.2.2 Type Distribution

Of the questions asked on our CS0 final exam, 11% of them were *why questions* (89% were *how questions*). Of these *why questions*, 75% of them were 32 and 25% of them were 23, meaning some kind of CS Speak was involved. Of the remaining sets, there are only three exams that have any *why questions*; CS1-Lopez, CS1-R1 and CS2-DCER. The distribution of *why questions* across these exams also never exceeds 15%. The remaining sets, CS0-Meerbaum-Salant, CS1-Lister and CS1-APCSA, have no *why questions* whatsoever. Although why questions might be considered challenging or subjective to grade, we hope instructors will consider the value of asking students to be able to explain their rationales and therefore situate their abilities and use of tools in the programming culture.

6. DISCUSSION

6.1 Culturally-Informed Learning Outcomes: *Why* Questions are Critical

Reflecting on the AT Taxonomy in light of situated cognition theory – the amalgamation of activities, tools and culture that define the programming community – we are struck by the relative scarcity of *why questions* among both our clicker and exam questions. Surely the integration of culture with activity and tools means that "just" being able to apply a tool or perform and activity is not sufficient. We wouldn't claim one was a vetted member of our community who couldn't explain *why* -- for any question in any of our 9 AT categories.

But *every clicker question is a why question* – regardless of whether the question's stem or options make that why explicit. It's inherent in the Peer Instruction pedagogy that questions are developed based on the kinds of discussions one wants the students to have (through the vote, discuss in small groups, revote procedure). Quite specifically, in Peer Instruction, the instructor frequently exhorts students to discuss not only why the right answers are right, but also why the wrong answers are wrong. Students aren't just "doing" an AT transition, but they are rationalizing their thought process and describing what they did and thought. In their groups, the class-wide discussion where they hear the explanations of other groups, and the instructor wrap-up, they are apprenticed in how programmers think about problems and rationalize actions.

We claim this means that we have identified 18 learning goals for development of programming students: All 9 categories have both how and why goals. So for example, by the end of the course students should be able to:

- In the context of a computational problem, read an English description and write code to solve the problem (13 how)
- In the context of a computational problem, read an English description and write code and describe why that code solves the problem in the English description. (13 why)

The truth is, as much as we do care about 13 how, in the absence of demonstrated 13 why ability, we cannot claim a student has, from a situated cognition perspective, become proficient as a practitioner in the community. Similarly though rarely a featured part of instruction, we need goals that students should be able to:

- Read code and give an English description of what it does (31 how)
- Read code and explain why their English description of what it does is correct (and possibly why another is not) (31 why)

These goals contribute to defining the community practitioner's ability to read the code of others, perhaps for code maintenance, modification, or debugging.

6.2 Can Summative Assessments Measure Computational Thinking?

We began this work with the goal of summarizing and elucidating our in-class learning materials. We hoped this would help us understand how a CS0 course which, at surface level, is focused on programming was achieving the positive changes in student confidence and abilities previous reported [2]. This paper has shown that a new view on how we want to think about, teach, and assess introductory programming courses can be found by stepping back and focusing on cognitive apprenticeship of computational thinking skills.

Interestingly, most of our identified 18 learning outcomes are not assessed on summative assessments – based on a sample selected primarily from recent research literature. Notably, *why questions* represent less that 15% of any exam – with three of the exams evidencing no *why* questions at all. Does this bother us as a community? We hope this work spurs discussion of that question.

Through our recent experiences in supporting high school teachers in the CS Principles project, the need to assess *why* ability has sharpened. We are beginning to explore multiple-choice questions (of the *how* variety) and asking students to explain in written English form how they analyzed it and why the wrong answers were wrong. These responses are more challenging to grade, but seem to be both more valuable and easier to create than multiple-choice *why questions* where the options are various explanations or rationalizations. In future work, we hope to further explore the potential and validity of these questions.

7. CONCLUSION

We propose that situated cognition theory, with its focus on learning for application in the "real world" and its focus on developing expertise within the context of a community is a useful lens for reconsidering programming instruction and perhaps computer science instruction more generally. A number of factors support this idea. First, our community has norms that seem particularly challenging for outsiders to understand. The common stereotype of a computer programmer makes it clear that we are to be considered other and our actions incomprehensible. It may be

that part of our collective recruitment and retention issues (at least in the US) stems from the lack of attention to acculturation of new members. The AT Taxonomy defines 18 learning outcomes that explicitly address the required culture, activity, and tools required for members of the programming community – which we believe needs to extend, at least minimally, to embrace everyone in modern, digital society.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF CNS-0938336 and NSF CNS-1138512. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] Brown, J. S., Collins, A. and Duguid, P. (1989) Situated Cognition and the Culture of Learning. *Educational Researcher* , Vol. 18, No. 1 (Jan. - Feb., 1989), pp. 32-42

[2] Cutts, Q., Esper, S., and Simon, B. 2011. Computing as the 4th "R": a general education approach to computing education. ICER 2011. ACM, New York, NY, USA, 133-138. DOI=10.1145/2016911.2016938 http://doi.acm.org/10.1145/2016911.2016938

[3] Ericsson, K. A. (2006). The influence of experience and deliberate practice on the development of superior expert performance. In K. A. Ericsson, N. Charness, P. Feltovich, and R. R. Hoffman, R. R. (Eds.). *Cambridge handbook of expertise and expert performance* (pp. 685-706). Cambridge, UK: Cambridge University Press.

[4] Collins, A., Brown, J. S., & Newman, S. E. (1990). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), Knowing, learning, and instruction: Essays in honor of Robert Glaser (pp. 453-494). Hillsdale, NJ: Lawrence Erlbaum.

[5] Wenger, E. (1998). *Communities of practice. Learning, meaning and identity.* New York: Cambridge University Press.

[6] Mazur, Eric. (2009). Farewell, Lecture? *Science.* Vol 323 No. 5910, pp. 50-51.

[7] Anderson, L.W., Krathwohl, D.R., Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J. and Wittrock, M.C. (eds.) (2001). A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives. Addison Wesley Longman.

[8] Johnson, C. G. and Fuller, U. (2006). Is Bloom's taxonomy appropriate for computer science? *Baltic Sea 2006*. ACM, New York, NY, USA, 120-123. DOI=10.1145/1315803.1315825 http://doi.acm.org/10.1145/1315803.1315825

[9] Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D., Riedesel, C. and Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bull.* 39, 4 (December 2007), 152-170. DOI=10.1145/1345375.1345438 http://doi.acm.org/10.1145/1345375.1345438

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

[11] Corney, M. W., Teague, D. M., Ahadi, A., & Lister, R, (2012) Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In de Raadt, Michael & Carbone, Angela (Eds.) *CRPIT*. Australian Computer Society Inc, RMIT University, Melbourne, VIC. (In Press)

[12] Falkner, K. and Palmer, E. (2009). Developing authentic problem solving skills in introductory computing classes. *SIGCSE 2009*. ACM, New York, NY, USA, 4-8. DOI=10.1145/1508865.1508871 http://doi.acm.org/10.1145/1508865.1508871

[13] Bareiss, R. and Radley, M. (2010). Coaching via cognitive apprenticeship. *SIGCSE 2010*. ACM, New York, NY, USA, 162-166. DOI=10.1145/1734263.1734319 http://doi.acm.org/10.1145/1734263.1734319

[14] McDowell, C., Werner, L., Bullock, H. and Fernald, J. "The Impact of Pair Programming on Student Performance of Computer Science Related Majors". (2003). *ICSE 2003*. Portland, Oregon.

[15] Braught, G., Martin Eby, L. and Wahls, T. (2008). The effects of pair-programming on individual programming skill. *SIGCSE 2008*. ACM, New York, NY, USA, 200-204. DOI=10.1145/1352135.1352207 http://doi.acm.org/10.1145/1352135.1352207

[16] Braught, G., Wahls, T., and Martin Eby, L., 2011. The Case for Pair Programming in the Computer Science Classroom. *Trans. Comput. Educ.* 11, 1, Article 2 (February 2011), 21 pages. DOI=1921607.1921609 http://doi.acm.org/1921607.1921609

[17] Gaillot, E., Bache, E. *CodingDojo*. http://codingdojo.org.

[18] Sanders, K., Richards, B., Mostrom, J., Almstrum, V, Edwards, S., Fincher, S., Gunion, K., Hall, M., Hanks, B., Lonergan, S., McCartney, R., Morrison, B., Spacco, J. & Thomas, L. (2008). DCER: Sharing Empirical Computer Science Education Data. *ICER 2008.* Sydney, Australia. 137-148.

[19] Meerbaum-Salant, O., Armoni, M. & Ben-Ari, M. (2010). Learning Computer Science Concepts with Scratch. *ICER 2010.* Rhode Island, USA. 69-75.

[20] Lopez, M., Whalley, J., Robbins, P., and Lister, R. 2008. Relationships between reading, tracing and writing skills in introductory programming. *ICER 2008*. ACM, New York, NY, USA, 101-112. DOI=10.1145/1404520.1404531 http://doi.acm.org/10.1145/1404520.1404531

[21] Lister, R., Simon, B., Thompson, E., Whalley, J., and Prasad, C. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ITICSE 2006*. ACM, New York, NY, USA, 118-122. DOI=10.1145/1140124.1140157 http://doi.acm.org/10.1145/1140124.1140157

[22] Whalley, J. L. Lister, R. (2009). The BRACElet 2009.1 (Wellington) specification. *ACE 2009*. Margaret Hamilton and Tony Clear (Eds.), Vol. 95. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 9-18.