Queensland University of Technology
Brisbane Australia

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

Teague, Donna M., Corney, Malcolm W., Fidge, Colin J., Roggenkamp, Michael G., Ahadi, Alireza, & Lister, Raymond (2012) Using neo-Piagetian theory, formative in-Class tests and think alouds to better understand student thinking : a preliminary report on computer programming. In *Proceedings of 2012 Australasian Association for Engineering Education (AAEE) Annual Conference*, Melbourne, Vic.

This file was downloaded from: `http://eprints.qut.edu.au/55828/`

© Copyright 2012 please consult the authors

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:*

Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming

Donna Teague[a], Mal Corney, Colin Fidge, Mike Roggenkamp, Al Ahadi[b] and Raymond Lister
Queensland University of Technology[a], University of Technology, Sydney[b]
Corresponding Author Email: Raymond.Lister@UTS.edu.au

BACKGROUND
Around the world, and for many years, students have struggled to learn to program computers. The reasons for this are poorly understood by their lecturers.

PURPOSE
When the intuitions of many skilled lecturers have failed to solve a pedagogical problem, then a systematic research programme is needed. We have implemented a research programme based on three elements: (1) a theory that provides an organising conceptual framework, (2) representative data on how the class performs on formative assessment tasks, and (3) microgenetic data from one-on-one think aloud sessions, to establish why students struggle with some of the formative tasks.

DESIGN / METHOD
We have adopted neo-Piagetian theory as our organising framework. We collect data by two methods. The first method is a series of small tests that we have students complete during lectures, at roughly two week intervals. These tests did not count toward the students' final grade, which affords us the opportunity to ask unusual questions that probe at the boundaries of student understanding. Think aloud sessions are the second data collection method, in which a small number of selected, volunteer students attempt problems similar to the problems in the in-class tests.

RESULTS
The results in this paper serve to illustrate our research programme rather than answer a single, tight research question. These illustrative results focus upon one very simple type of programming question that was put to students, very early in their first programming subject. That simple question required students to write code to swap the values in two variables (e.g., `temp = a; a = b; b = temp`). The common intuition among programming lecturers is that students should be able to easily solve such a problem by, say, week 4 of semester. On the contrary, we found that 40% of students in a class at one of the participating institutions answered this question incorrectly in week 4 of semester.

CONCLUSIONS
What is emerging from this research programme is evidence for three different ways in which students reason about programming, which correspond to the first three neo-Piagetian stages (Lister, 2011). In the lowest and least sophisticated stage, known as the sensorimotor stage, novices exhibit two types of problems: (1) misconceptions that are already well known in the literature on novice programmers (e.g., Du Boulay, 1989),and/or (2) an approach to manually executing ("tracing") code that is poorly organized and thus error prone. Novices at the next stage, known as the preoperational stage, can correctly trace code, but they cannot reliably reason about a program in terms of abstractions of the code (e.g., diagrams). It is only at the third stage, the concrete operational stage, where students begin to exhibit some capacity to reason about code abstractions. However, traditional approaches to teaching programming implicitly assume that students begin at the concrete operational stage.

KEYWORDS
Programming, research, neo-Piagetian, think aloud.

Introduction

It is acknowledged around the world that many university students struggle with learning to program (McCracken et al., 2001; McGettrick et al., 2005). In this paper, we describe how we have developed a research programme to systematically study and incrementally improve our teaching. We have adopted a research programme with three elements: (1) a theory that provides an organising framework for defining the type of phenomena and data of interest, (2) data on how the class as a whole performs on formative assessment tasks that are framed from within the organising framework, and (3) data from one-on-one think aloud sessions, to establish why students struggle with some of those in-class formative assessment tasks. We teach introductory computer programming, but this three-element structure of our research is applicable to many areas of engineering education research.

Design: Neo-Piagetian Theory of Cognitive Development

We have adopted neo-Piagetian theory as our organising framework (Morra, Gobbo, Marini, & Sheese, 2007). It is a derivative of classical Piagetian theory. Classical Piagetian theory focuses on a child's development of general abstract reasoning skills as they grow older. Neo-Piagetian theory instead focusses on people of any age as they acquire increasingly abstract forms of reasoning within a specific problem domain. In neo-Piagetian theory, a person may display high abstract reasoning abilities in one domain but not in an unrelated domain. Both the classical and neo-Piagetian theories define four stages of cognitive development which, from least mature to most mature, are: Sensorimotor, Preoperational, Concrete Operational and Formal Operational. These four stages are described below.

The sensorimotor stage is the first stage of cognitive development. At this stage, the novice does not possess or cannot appropriately apply the abstractions needed to reason about a particular problem. For example, a novice who is at the sensorimotor stage in Newtonian mechanics might in fact reason about a mechanics problem in Aristotelian terms, or might not be aware of the appropriate Newtonian concepts to apply. In the context of programming, Lister (2011) defined the sensorimotor stage as being exhibited by "students who trace code with less than 50% accuracy" (where 'trace' refers to the act of manually executing a piece of code with specific initial values, to derive the final values in the variables). A novice at this stage of development in programming can only trace code with considerable effort and for this reason such a novice is often disinclined to make use of tracing to solve programming problems. When such a novice traces code to find a bug, he/she tends to use ad hoc initial variable values, rather than values deliberately chosen to help identify a bug.

The preoperational stage is next. At this stage, the novice does possess some appropriate abstractions, but these abstractions are fragmented and do not make up a coherent understanding of the problem domain. Such a novice might be able to use a concept when specifically told to do so, but may not spontaneously apply that same concept when it is appropriate but the novice is not explicitly told to do so, or the preoperational novice may apply that concept when (to the expert) it is egregiously inappropriate to do so. In the context of programming, according to Lister (2011), preoperational students can trace code, but they tend not to abstract from the code to see a meaningful computation performed by that code. The preoperational programmer will, for example, struggle to make effective use of the relationship between code and a diagram that represents the function of that code.

The concrete operational stage follows the preoperational stage. This is the first stage where the novice can routinely reason about abstractions of code. However, this 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.

The formal operational stage is the ultimate stage of Piagetian reasoning. It is the level at which the expert performs. A person thinking at the formal operational stage can reason logically, consistently and systematically. Formal operational reasoning also requires a reflective capacity - the ability to think about one's own thinking within the given problem domain. As this neo-Piagetian stage is expert thinking, and not what we see or test for in our students, we will not discuss this stage any further in this paper.

Method

In-Class Formative Tests

We collected in-class test data from students in our lectures. We conduct such tests at intervals of approximately two weeks, throughout semester. In the most recent semester, we conducted 9 in-class tests. In each in-class test, students were given a single printed sheet (usually double-sided) containing several short tasks.

In this section we illustrate our use of in-class tests by presenting the second of nine tests from the most recent semester. That test, which students completed at week 4, is shown in Figure 1. Unlike other tests, this test fitted on one side of the sheet, which is why (because of space limitations) this test was chosen as the illustration. In Figure 1, the original test has been annotated, to show: (1) the number of students who did the test (i.e. 105); (2) sample solutions to the questions (shown in the boxes where students would have handwritten their own answers); and (3) the percentage of students who answered each question incorrectly. This annotated version of the test was actually given to students as feedback.

The first question in Figure 1 is intended as a screening question. Its purpose was to determine if students understood the fundamentals of variable declaration and assignment. A student who has trouble answering this very simple question is probably operating at the sensorimotor stage. By this week of semester, most lecturers teaching programming would expect that their students could answer this question.

Question 2 requires students to write code to swap values stored in two variables. Students had been given the same question, but with different variable names, in their first test a week earlier. After that first test, the lecturer had discussed the solution to that question. A student who could answer Question 1 in this second test but not answer Question 2 may be operating at the preoperational stage. However, students who answer Question 2 correctly are not necessarily preoperational, as they may answer this question from memory.

The last question on the test is an extension of the swapping question and requires code to 'rotate' the values in four variables. Students had not seen this question before. The intention of Question 3 was to see if students who answered Question 2 correctly could then transfer their thinking to Question 3. A student who can answer one of these two questions, but not both, is probably operating at the preoperational stage.

Think Alouds

When analysing student answers to in-class tests, many assumptions can be made about how a student arrived at a particular answer. In order to extract more reliable information from students about their reasoning when solving in-class tests, weekly one-on-one think aloud sessions were conducted with a small number of volunteer students.

Ericsson & Simon (1993) advocate think aloud verbal reports as an effective means of data collection. Atman and Bursic (1998) also found that, when studying the processes of students performing engineering design tasks, they were better able to discern mental processes from verbal reports rather than from the final products of a design process.

Name:______________________________Student #:___________________Date: 22/ 3 /2012

Semester of learning to program: ___1st ____ Week in current semester:___ 4 ___

Q1 In the boxes, write the values in the variables after the following code has been executed:

```
int a = 23;
int b = 11;
int c = 61;

a = b;
c = a;
b = c;
```

105 students submitted this test.

17% got it wrong

The value in a is 11 the value in b is 11 and the value in c is 11

Q2 Assume the integer variables black and white have been initialised in Java. Write code to swap the values stored in black and white.

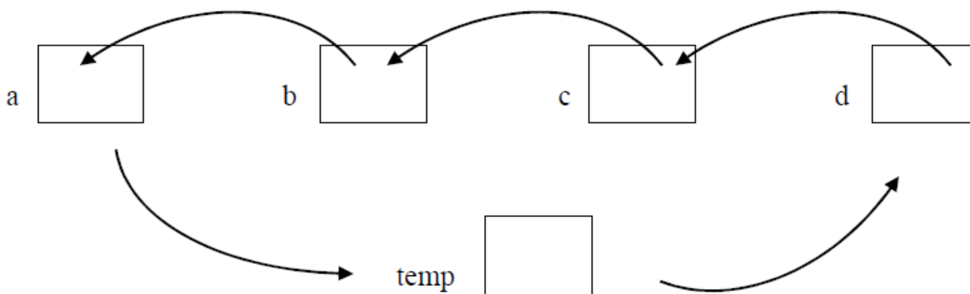
40% got it wrong

```
temp = black;          -- or --     temp = white;
black = white;                      white = black;
white = temp;                       black = temp;
```

Other names than "temp" are fine, but use a meaningful name.

Q3 Suppose there are four variables, a, b, c and d as depicted below:

Write code to move the values stored in those variables to the *left*, with the **leftmost** element being moved to the **rightmost** position - as depicted by this diagram:



For example, if initially a=1, b=2, c=3 and d=4, then after your code is executed those variables should contain a=2, b=3, c=4 and d=1. (But your code should work for all possible values.)

```
temp = a;
   a = b;
   b = c;
   c = d;
   d = temp;
```

36% got it wrong

Other names than "temp" are fine, but use a meaningful name.

Figure 1: The in-class formative test used in week 4 of the semester.

In the think aloud sessions, the students were not asked to explain or describe what they were doing. To ask them to do so might distract them from solving the task. Instead, as advocated by Ericsson & Simon, students were asked to simply verbalise the information they were attending to as they performed the task.

Our use of think aloud sessions is an example of the microgenetic research method, which is defined as having three main properties (Siegler, 2006, p. 469): (1) Observations span the period of rapidly changing competence (i.e. a semester), (2) the density of observations is high, relative to the rate of change (e.g. weekly think alouds), and (3) observations are analysed intensively, to infer the representations and processes used by the students.

Each think aloud session was digitally captured using a Smartpen and dot paper (LiveScribe, 2012). The scripts from these sessions were then processed to produce "pencast" PDFs, the audio-synced video contents of which are re-playable using Adobe Acrobat Reader.

RESULTS

In-Class Formative Tests

Table 1 shows a contingency table for Questions 1 and 2. Twelve students answered both questions incorrectly, and those students are likely to have been operating at the sensorimotor stage. Fifty seven students answered both questions correctly, and those students are likely to be reasoning at least at the preoperational stage. It is harder to classify the 36 students who answered one of these questions correctly, but not both. However, 30 of those students (83%) answered Question 2 incorrectly (even though these students had encountered a swapping problem the previous week), which offers statistical support to the neo-Piagetian hypothesis that the ability to conduct the sensorimotor task of Question 1 is an earlier developmental stage than the preoperational task in Question 2.

		No. students whose answer to Q2 was:	
		incorrect	correct
No. students whose answer to Q1 was:	incorrect	12	6
	correct	30	57

Table 1: A Contingency Table for Questions 1 and 2. (n = 105. $\chi 2$ test, $p = 0.01$)

Table 2 shows a contingency table for Questions 2 and 3 from the in-class test in Figure 1, but only for the 87 students who answered correctly the screening question (i.e. Question 1). Of these 87 students, 66 (i.e. 76%) answered questions 2 and 3 consistently. That is, they either answered both questions incorrectly or both questions correctly. The 18 students who answered both questions incorrectly are likely to be preoperational, while the 48 students who answered both questions correctly are likely to be concrete operational. It is harder to classify the remaining 21 students (24%) who answered questions 2 and 3 inconsistently, although their inconsistency does suggest they are reasoning preoperationally.

		No. students whose answer to Q2 was:	
		incorrect	correct
No. students whose answer to Q3 was:	incorrect	18	9
	correct	12	48

Table 2: A Contingency Table for Questions 2 and 3. (n = 87. $\chi 2$ test, $p < 0.001$)

Clearly, it is not possible to draw many firm conclusions from a single in-class test (apart from that a surprising percentage of students struggle on what many lecturers would consider to be trivial problems). Recall, however, that we conducted nine tests of this general nature over the duration of our most recent semester. When the data across these tests is combined, patterns can be seen, but space limitations do not allow elaboration on those patterns. For some further results on in-class tests, we refer the reader to Corney et al. (2012). As discussed in an earlier section, our purpose in presenting the results for this in-class test is to illustrate the type of research programme we are running. Our aim in this paper is not to analyse in-class data in detail, but instead to illustrate how we use a combination of in-class tests and think aloud data in our ongoing research programme.

A Think Aloud Session with Bobcat

We now describe a think aloud session with a student, which demonstrates why some students struggled with the 'swap' problem in the in-class test. This student calls himself 'Bobcat' in think aloud sessions, which is not his real name. He is enrolled at a different university from where the above in-class test was done. However, this second university is also part of our research programme, and we are seeing similar results from in-class tests at both universities. At Bobcat's university, students are taught programming in the Python language, whereas the students at the other university are learning Java, but the swapping problem is so simple it is almost identical in both languages.

At the time of this think aloud session, Bobcat was in his fifth week of his first programming subject. He had seen the 'swap' problem twice before (but with different variable names). In an in-class test in week 2, but he did not even attempt that problem. The lecturer reviewed and explained the solutions for the week 2 test in that same lecture. A similar 'swap' problem was then given in the week 4 in-class test. This second time, Bobcat attempted the exercise, but provided the following egregiously incorrect solution:

```
first = temp
second = first
```

Bobcat's solution shows that he was aware of the need for a temporary storage variable (although perhaps he did so from his memory of the week 2 solution), but he demonstrates a poor understanding of the swapping process. From his answer to this week 4 in-class test, we can only guess what Bobcat was thinking. The following think aloud session with Bobcat illustrates how think aloud data complements and enhances the in-class data.

Bobcat starts his think aloud session by slowly and carefully reading the question, twice. He mentions needing a 'temp file', which we interpret as meaning a temporary *variable*. He then writes the first line shown in Figure 2, which correctly assigns the value of `first` to a third variable which he calls '`temp`'. He then says:

Line 1 temp = first
Line 2 Second =

Figure 2: Bobcat - 1st Attempt

Now that's stored away, I can say that ... how am I going to do that?

Bobcat repeats aloud the one line of code he has written so far, "`temp` equals `first`". There is a pause in the pencast before he says "I can say `second` equals..." and he then starts to write code to assign some value to `second` as shown in Figure 2 at Line 2. There is another long pause after which he says "... Now, I've got myself confused here. No."

Bobcat re-reads the code he has written, adding that "`temp` is stored away. Second equals <pause> I know I can write this out". Then he says:

Yeah, I know that `temp`'s got to get stored away for a `temp` file...to swap them around. It's just ... why am I being confused?

Before writing anything else, Bobcat considers what should be assigned to `second`.

> *If `second` now equals `first`, ... how am I going to get `first` to equal `second`?*

Then he mutters "`second` equals `temp`". He changes his mind and says "`second` equals `first`, so that swaps ... so that will mean ..." After another pause, Bobcat is asked by the interviewer to explain the purpose of the first line of his code (that is, where he assigns `first` to `temp`). Before he answers, Bobcat says that perhaps that first line of code should have actually been assigning `second`'s value (not `first`'s) to `temp`. (In fact, If Bobcat had a good grasp of swapping, he would know it does not matter whether the first line is *`temp = first`* or *`temp = second`*.)

Again, an intervention by the interviewer prompts for an explanation of his first line of code. Bobcat quickly explains it "puts `first`'s value into a `temp` file so it can't get changed", thus clarifying that he believes it is `first`'s value, not `temp`'s value, that is being stored. He further explains:

> *If I had said `first` equals `second`, and `second` equals `first` I'm going to lose the value of `second`. No I'm going to lose the value of `first`. I know how to do this - that's the thing!*

Given the extent to which he is struggling, it is surprising that he thinks "*I know how to do this".* From our experience with other students in think alouds, who have made a similar claim, we suspect these students tend to regard programming as a process of recall.

At this stage, almost 4 minutes have elapsed since Bobcat started. Bobcat starts reading his code again, explaining it to himself as he goes. Pausing after Line 1, he decides:

> *No, `temp` should equal `second`. I don't know. I'm really lost in this! I know how to do this. That's what gets me, I get frustrated and I'm gone!*

Again, the interviewer intervened to ask about the purpose of storing `first`'s value at Line 1. Again, Bobcat explained the need to use a temporary variable in order to swap the values of two other variables. The interviewer then provided a prompt to continue that line of reasoning: "Ok so you've stored `first`, ...?"

Bobcat now says something that offers a clue to the source of his confusion:

> *If I say `first` equals `second`, that's going to ...change `second` and the `temp` file ... because `temp` ... has the value of `first`.*

What Bobcast articulated above is a well known novice misconception of how the assignment operator "=" works (Du Boulay, 1989). Bobcat thinks (at least some of the time) that assignment is like the mathematical symbol "=", so that (for example) `temp = first` binds the two variables together, so that that the value in the two variables will always be the same, and that a subsequent `second = first` will bind all three variables together.

After six minutes had elapsed, Bobcat decided to write down a complete attempt at the second line of code. He crossed out his incomplete attempt at the second line and replaced it with what is shown at Line 2 in Figure 3. He then continued almost immediately to write the third line shown in Figure 3. Those three lines form a correct solution.

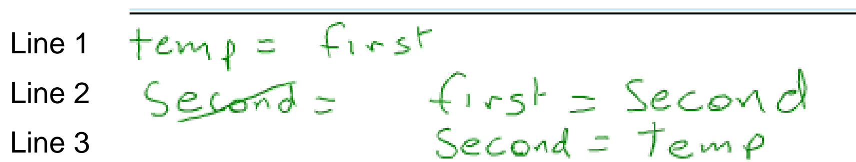


Figure 3: Bobcat - 2nd Attempt

If Bobcat had written the solution in Figure 3 as his answer to an in-class test, that answer would have been marked as correct and we might have incorrectly assumed that he had a good understanding of the task. The think aloud data indicates he does not understand, especially when he adds:

> *if I say* `first` *equals* `second`*, why doesn't that change the value of* `temp`*, if* `temp` *equals* `first`*? Now why doesn't* `temp` *equal* `second`*? ... And that's where I'm lost.*

The interviewer then asks Bobcat to trace his code, using initial values of his choosing. His attempt at doing this at first seems misguided, as he starts to record the value of `first` in multiple places, as shown in Figure 4 at Line 4. However, after prompting about the type of notation to use, Bobcat completes the trace and concludes that his code does indeed swap the two variables. It took Bobcat a total of eight and a half minutes to reach this point.

Line 1
Line 2
Line 3
Line 4

Figure 4: Bobcat - Trace

By articulating at one time his misconception about the assignment operator, but then at a later time demonstrating that was able to trace his code, Bobcat illustrates one of the features of novice reasoning at the sensorimotor and preoperational stages − over a period of time, their reasoning is often not consistent.

We would not with certainty place any student into a neo-Piagetian category based upon a single think aloud task. In this particular task, Bobcat displays aspects of both the sensorimotor and preoperational stages – he might possibly be regarded as someone in transition from sensorimotor to preoperational. While he does articulate a misconception about how variable assignment works (which is evidence of the sensorimotor stage), he is able to trace code (albeit with some prompting). He can explain why there must be a third variable, which is evidence of reasoning at the preoperational stage. His eight and a half minute struggle to write three lines of code for a swap firmly establishes that he is not yet at the concrete operational stage.

For some further results on think aloud data, on subjects other than Bobcat, we refer the reader to Teague, Corney, Ahadi & Lister (2013).

Conclusions

The work we describe in this paper is preliminary and ongoing, but our research approach is already giving us a better understanding of why many of our students struggle with learning to program. In the traditional introductory programming lecture, the PowerPoint slides are peppered with diagrams, in the belief that these abstractions help all students. These abstractions probably do help the students who are reasoning at the concrete operational stage. However, we now see that these abstractions are of little use to students who are reasoning at the sensorimotor or preoperational stages. We now see that different methods of teaching are required for students at the sensorimotor and preoperational stages. Sensorimotor students need help correcting misconceptions and help with learning how to systematically and reliably trace code. Preoperational students need help with seeing abstractions of code.

Irrespective of exactly what conclusions can be drawn from this preliminary report, this paper serves to illustrate an approach to pedagogical change that applies to engineering education research in general. We believe that progress on difficult pedagogical problems is slow. When a particular aspect of pedagogy has resisted the intuitions of many skilled lecturers, over many years, then a single paper, describing a single experiment, is unlikely to deliver the solution. Instead the solution to a difficult pedagogical problem will usually come from first putting in place a research programme and then making incremental changes over several years, on the basis of research data that is collected and analysed routinely.

References

Atman, C. J., & Bursic, K. M. (1998). Verbal Protocol Analysis as a Method to Document Engineering Student Design Processes. *Journal of Engineering Education, 87*(2), 121-132.

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Paper presented at the Australasian Computing Education Conference (ACE 2012).

Du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* (pp. 283-300). Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.

Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the Australasian Computer Education Conference (ACE 2011).

LiveScribe. (2012). Retrieved July 10, 2012, from http://www.smartpen.com.au/

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). ITiCSE 2001 working group reports: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-140.

McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Loverove, L., & Mander, K. (2005). Grand Challenges in Computing: Education - A Summary. *The Computer Journal, 48*, 42-48.

Morra, S., Gobbo, C., Marini, Z., & Sheese, R. (2007). Cognitive Development: Neo-Piagetian Perspectives. *Psychology Press*.

Siegler, R. S. (2006). Microgenetic analyses of learning. In W. Damon & R. M. Lerner (Series Eds.) & D. Kuhn & R. S. Siegler (Vol. Eds.) *Handbook of Child Psychology (6th ed)* (Vol. 2: Cognition, Perception and Language, pp. 464-510). Hoboken, NJ: Wiley.

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013). *A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers*. Paper presented at the Australasian Computing Education Conference (ACE 2013).