AP CS Principles Pilot at University of California, San Diego

Authors: Beth Simon (UCSD) and Quintin Cutts (University of Glasgow)
Course Name: CSE3 Fluency with Information Technology
Pilot: Fall 2010 quarter and Winter 2011 quarter

a) How CSE3 Fits In At UCSD:

UCSD's Computer Science and Engineering department has offered CSE3, *Fluency with Information Technology* for a decade. In the past 5 years or so, it has been a required course for two groups on campus: psychology majors and students enrolled in UCSD's Sixth College undergraduate program. At UCSD all students have their general education requirements set through their choice of a "college"; and Sixth College's focus is on culture, art, and technology. The Pilot offerings in Fall 2010 and Winter 2011 served ~1000 students. Sixth students (primarily freshmen) accounted for 70% of the population and were spread across all majors except engineering. Psychology majors were primarily juniors or seniors. The course was ~60% women. The course content of CSE3 had been under evaluation for a year when Dr. Simon was presented with the opportunity to pilot CS Principles. Upon review, the newly recommended CSE3 curriculum appeared to fulfill the grand majority of CS Principles learning goals and, hence, seemed a great fit.

b) Course Stats

- **Lectures**: 2 80-minute periods, TTh
- **Labs**: 1 120-minute closed lab with TA instruction, spread over the week
- **Course**: 10-week quarter, yielding 50 contact hours
- **Credit Hours**: 4
- **Fulfills Requirements**: General Ed for Sixth College, Psychology Major
- **Attendance**: Fall 2010: 580 started; 572 finished. Winter 2011: 456 started; 447 finished
- **Programming Language**: Alice followed by Excel
- **Grading**: daily quizzes, daily peer instruction/clicker questions, 9 lab exercises, individual programming project, midterm, final
- **Pilot**: Fall quarter, 2010, Winter quarter 2011

c) Course Design

The course was designed with the following key underlying questions: If this were the last course someone was to take in computing, what would you want them to know? What do you want them to get out of it? Our answer to this was to help students develop not only the knowledge, but the visceral understanding that computers are:

1. Deterministic – they do what you tell them to do

2. Precise – they do *exactly* what you tell them to do, and
3. Comprehensible – the operation of computers can be analyzed and understood.

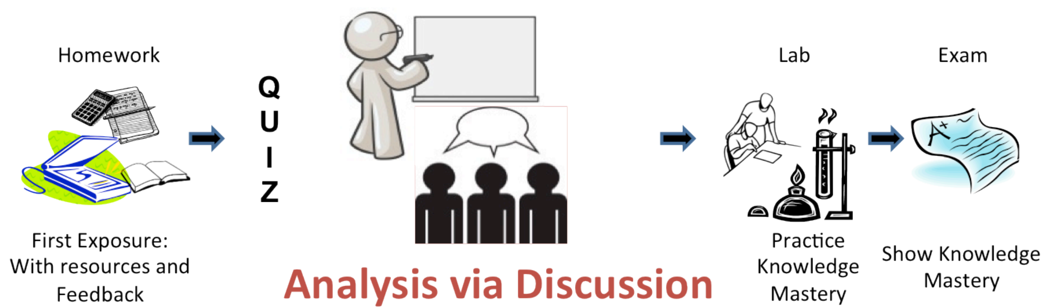
As such, we agreed with the CS Principles belief that "programming" is one of the best tools we have for helping students get at the *heart* of computing – that you can provide directions to the computer and it will do what you say. But our previous experiences led us to believe that simply "getting students to program" didn't effectively convey that message to the masses. We'd seen all too many students leave introductory programming experiences not with a sense of control, but with a more firmly developed sense of mystery, that computers can't be understood and you just move things around and change stuff until things finally do what you want.

We took as our mantra to students that we didn't care so much if they could write a program to do something – if they couldn't also explain how it worked and be able to analyze what it did and why. Analysis and communication – not program writing – was our goal for students. To support them in developing these skills, we adopted course design based in the Peer Instruction pedagogy [4]. In a nutshell, Peer Instruction recognizes that it is students who need to do the learning (not the professor) and provides the structure and time for students to spend in analysis and discussion of challenging questions that engage them in confronting difficult concepts. The standard course "lecture" is replaced by a two-part process:

1) Preparatory work outside of lecture (pre-reading of the textbook, though in this class augmented by work in Alice building the programs described in the text) and
2) Replacement of standard "demoing" or explanations with a series of multiple-choice questions where students are engaged in analyzing a problem or some code, and describing what the code does or selecting some code to make a specific thing happen.

The preparatory work involving program building derives from our awareness of students' need for real experience of programming concepts and constructs if they are going to have meaningful discussions about what programs mean. Simply reading about programming doesn't give the necessary depth of experience or engagement – learners need to actually see the constructs working.

The in-class multiple-choice questions are asked using a specific format. First, each student attempts the question him/her-self (and indicates their answer electronically with a clicker). Then, students spend a few minutes discussing the question with a pre-assigned group of 3 students. The groups are exhorted not just to agree on the answer they think is correct, but to also explain their thinking to each other and to discuss why the wrong answers are wrong. Finally, a second round of answers is collected by clicker, and a class-wide discussion is led where students are asked to share the thinking and ideas discussed in their groups. The instructor can augment these explanations by modeling how she thinks about the problem and clarifying any specific misunderstandings that might persist.



Schematic diagram of the peer-teaching approach.

Final assessment of understanding is measured through weekly 2-hour closed labs (covering the previous week's materials) and standard exams (mostly multiple-choice questions, with some written explanations and code writing required).

In addition, students completed both a 4-week individual project in Alice to create a digital animation or video game of social value (communicating their views on a subject of import to society, or providing educational value). The course goals were also supported by three "Technology and Society" assignments that sought to focus students on how the programming concepts they were learning existed in the world around them. These also engaged students in digital communication techniques such as discussion forum posting and wiki creation and editing. The way in which these additional assignments were able to link the students' programming work with their everyday lives sought to give *meaning* to the whole process.

The course content was derived from *Learning to Program in Alice* (Dann, Cooper, Pausch) Chapters 1,2 (sequential execution), 4 (methods, parameters), 5 (events), 6 (functions and if statements), 7 (loops), and 9 (arrays/lists); and a popular Excel book (basic formulas and functions, managing large datasets, exploring data with PivotTables).

d) **Evidence of Student Success/Valuation**

As a general education course, one important metric of success is pass rate for the course. Yes, this course needs to be rigorous and provide students the training and understanding to be confident and capable in using technology in their lives. However, students also need to be able to succeed in the course without lengthening their time to degree. Given high failure rates in standard introductory computing courses, this is a non-trivial issue. The pass rate for the fall term was 98% and the pass rate for the winter term was 97%.

But did we succeed in meeting our educational goals? To help measure this, we sought answers to questions like: "What if this is the last computing course these students ever take? What are they getting out of it? Does this satisfy us with regards to what an informed populace should know?" We were teaching "programming" but our lectures focused students on analyzing programs to illuminate core concepts and skills. We asked students about their experience in lab during Week 8:

Learning computing concepts may have opened many doors for you in your future work. Although you may not ever use Alice again, some of the concepts you have learned may become useful to you. Some examples include:

- *Understanding that software applications sometimes don't do what you expect, and being able to figure out how to make it do what you want.*
- *Being able to simulate large data sets to gain a deeper understanding of the effects of the data.*
- *Understanding how software works and being able to learn any new software application with more ease, i.e. Photoshop, Office, MovieMaker, etc.*

Aside from the examples given, or enhancing the examples given, please describe a situation in which you think the computing concepts you have learned will help you in the future.

A more complete analysis of students' answers can be found in [5], however we found seven categories of response types among students' responses, with students reporting on average at least two of these each:

- Transfer, near: can apply new skills in software use
- Transfer, far: can use problem solving skills in other areas of life
- Personal Problem Solving Ability: Debugging – can logic it out, attempt to, or deal with, unexpected behavior
- Personal Problem Solving Ability: Problem Design – can develop plan to solve technical problem, can see what requirements exist
- View of Technology: greater appreciation or understand of technology
- Confidence: increased ability to do things on computer, a can-do attitude
- Communication: communicate better about technology

The student answers themselves convinced us that the class had had a notable impact on many students' lives. As an example, one student answered:

"*The things I learned in Alice can help me not to be so frightened in general when dealing with technology. Although I am not certain I have absolutely mastered every concept in Alice, I am certain that I have learned enough to bring me confidence to apply these ideas in the technological world. This is a big deal for me, as I do consider myself quite technologically challenged. I think this class has given me tools for life, that can be applied to both my life at home, socially, and at work.*"

e) **What Worked And Didn't**

The Language. Worked. The drag-and-drop nature of Alice did provide students the reduced frustration and cognitive load in developing programs – enough so that we were able to ask students to "develop" programs on their pre-class preparation, without them getting stuck half-way. However, just as important was the fact that the execution model of an Alice program is "visible" to students – that is, for every instruction they create they can "see" the effect of that instruction in execution of

their program. (Note: we didn't cover variables in our seven weeks of instruction – somewhat a shock to us as instructors initially, but, in the end, a critical recommendation from Steve Cooper). In group discussions in class it was clear how students would match up observed behavior and instructions – and how they came to understand that debugging is a comprehensible process of matching code with outcomes.

Peer Instruction. Worked. Peer Instruction's core premise is to provide students the opportunity and excuse to engage with deep challenging concepts. Originally developed in physics to help address the fact that students could "plug and chug" their way to correct answers without really understanding applicable physical laws, Peer Instruction was an excellent fit for the goals of this course. Programming is a great tool for engaging students, because they can see for themselves if they have "got it right". However, that's also a potential pitfall. We used Peer Instruction to keep students focused on the goal of understanding how programs and programming concepts worked – not just getting a program to do what you want.

Excel. Worked. It was surprisingly rewarding to cover "basic" Excel topics *after* having done Alice. Students were somewhat amazed to see how previous experiences with Excel could now be seen in a new light. Students saw how a function like sum was, well, a function. It takes parameters and returns a number value. Nested if statements in Excel were equally easy. Students could identify (on the exam) that VLOOKUP is a loop over an array of items with an if-statement inside.

Focus on Analysis and Communication. Needed improvement. The first term of the pilot we did not make clear enough to students (from the beginning) that our goal was to help them develop technical analysis and communication skills, both to be able to interact with computers better themselves, and to be able to interact more effectively with computing professionals. Students are spending a lot of time programming in Alice, and it's important to emphasize almost daily how learning to use Alice is just a means to an end – the end being an understanding that computers are deterministic, precise, and comprehensible.

Multiple Classrooms via Video. Didn't work. For reasons outside the scope of this pilot, each term the class was offered simultaneously in three rooms co-located in one building. The instructor "lectured" in one room and video of her was streamed live to the adjoining rooms. Although all rooms were staffed with actively engaged tutors and TAs (e.g. as support during discussion periods) and that the instructor walked among all the rooms during most every discussion period, students made it clear that they didn't like this "video" format – though the concern was primarily among freshmen; upper division students felt that the style of the classroom (using Peer Instruction) was all that really mattered.

f) **Links, Resources, Acknowledgments**

The pilot was taught by Beth Simon, but collaboratively developed and inestimably improved by Quintin Cutts, visiting on sabbatical from the University of Glasgow.

Extreme thanks go to our intrepid army of TAs and undergraduate tutors (all 40+ of them), without whom the course could not have been the same.

A web site for educators interested in adopting our curriculum: http://www.ce21sandiego.org/

© ACM, 2012. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Inroads, {VOL 3, ISS 2, (June 2012)} http://doi.acm.org/10.1145/2189835.2189854