Experience Report: an AP CS Principles University Pilot

Beth Simon, Sarah Esper
Computer Science and Engr. Dept.
University of California, San Diego
La Jolla, CA USA
+1 858 534 5419
{bsimon, sesper}@cs.ucsd.edu

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

ABSTRACT

We report on the development and deployment of a pilot of the new Advanced Placement CS Principles course in the United States. The course is designed to introduce core computational concepts and instill computational thinking practices. We report on an initial offering with 571 university students, mostly non-CS majors taking the course as a general education requirement. We discuss the instructional design supporting the course, describe how the various components were implemented, and review student work and valuation of the course. Though the course appears to "teach programming" in Alice, students reported gaining significant analysis and communication skills they could use in their daily life. We reflect on how instructional design decisions are likely to have supported this experience and consider the implications for other K-12 computing/IT education efforts as well as for regular CS1 courses.

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

General Terms
Human Factors

Keywords
CS0, peer instruction, clickers, Alice, computational thinking.

1. INTRODUCTION

There is a world-wide effort to expand computing education and specifically computational thinking skills to secondary schooling. In the 20th century, the three competencies deemed essential for the development of an educated populace capable of meeting societal needs were reading, writing and arithmetic. With the penetration of ever cheaper and more powerful computers, 21st century competencies will need to include understanding of the basics of how computers work and the power and processes of computation. Future professionals in any arena will need significant analytical and communication skills to support their workplace technology use. Here, we report on a course designed to meet these needs. Specifically, we describe a) the instructional design and implementation and b) the student experience of a general-education computing course at the University of California, San Diego.

The *instructional design* of this course is critical. The usual designs for computing course must be reconsidered in light of:

- a different student population (e.g., a required course for (potentially) non-interested students)
- markedly different desired learning outcomes (e.g., not preparing students for a specific next course, but providing students with skills necessary to support their life-long needs)

We view *student experience* in this course as more important than typical narrow definitions of student success. Our metric is not simply "can they write programs." Rather, we seek to know how their experience of reading, analyzing, writing, debugging and discussing programs have *changed* them. How are they prepared to understand, work effectively with, and continue to expand their computational experiences? From this pilot offering, we report ad-hoc analysis with the intention of describing the breadth and variation of student experiences. Rigorous analysis is beyond the scope of this paper, and is the subject of future work.

2. BACKGROUND

CSE3: Fluency in Information Technology is a UCSD course developed in 2002, originally following Larry Snyder's text of the same name. It serves as a general education requirement for ~800 UCSD undergraduates per year, and also a departmental requirement for psychology majors (~300 per year). In 2008 a multi-departmental committee reviewed the curriculum resulting in recommendations to focus on a) computational concepts (over skills) and b) problem-solving, specifically employing programming with a contextualized approach. A quarter-term syllabus was proposed featuring 7 weeks of Alice and 2 weeks of Excel content (with 1 week of open topics).

Concurrently, Dr. Simon was invited to pilot a proposed Advanced Placement CS Principles course. CSE3's syllabus provided significant coverage of CS Principles learning goals. During the development of materials, Dr. Simon mostly followed the UCSD committee's recommendations, but kept the six CS Principles computational thinking practices firmly in mind. They are:

1. Analyzing effects of computation
2. Creating computational artifacts
3. Using abstractions and models
4. Analyzing problems and artifacts
5. Communicating processes and results
6. Working effectively in teams

3. INSTRUCTIONAL DESIGN

The instructional design of the course was strongly influenced by logistical factors including: the need to deliver the course to large numbers of students per term (>500), the existing 2-hour "closed" lab component, and the common use of many undergraduate tutors in support of the course. Although the course was taught simultaneously in three rooms by video feed, this wasn't a strong factor in course design. The design was also critically and positively influenced by the arrival of Dr. Quintin Cutts as a sabbatical visiting researcher. Dr. Cutts' research interests include PI, learning to program, and the development of school computing curricula.

3.1 Overarching Needs/Decisions

The course is designed around Peer Instruction (PI), an instructional technique well-documented to show increased learning. The most noted research has occurred in physics where both a large study of 6,000 students and a controlled study by Eric Mazur showed at least a 2-fold increase in student learning when PI was used in place of a "standard" lecture [1, 2]. Dr. Simon's success in adopting PI in introductory computing [5] prompted its adoption here. Crucially, PI engages the students in lecture – often to the extent where they spend 75% of class time in the process of problem solving.

In the standard PI model, before class, students gain preparatory knowledge (e.g. through reading the textbook) and complete a pre-lecture quiz on the material. During class, lecture can be interspersed with or largely replaced by a series of multiple choice questions (MCQs) designed to engage students in deepening their understanding of the material. MCQs often focus on deep conceptual issues or common student misconceptions or problems. This is instantiated via a 4-part process:

1. Students individually consider a question and select an answer (typically reporting it via use of a clicker).
2. Students discuss in pre-assigned groups (note: they have not been shown the class-wide results of the first "vote").
3. Students vote again on the same question.
4. Class-wide discussion follows led by student explanations and instructor lecture/modeling of the problem solving process.

Dr. Cutts raised concern over the pre-reading and quiz not adequately preparing students for lecture and Dr. Simon agreed that her previous experiences in developing reading quiz questions devolved into factual recall questions. At issue is the need for students to be engaged with programming concepts and this is unlikely to happen with reading alone. In response, we developed new "exploratory homework" that leverages Alice's support for "learn by playing". This is described in detail below.

Another pedagogical goal was to provide opportunity for spaced (rather than massed) learning of concepts [4]. Research in educational psychology has clearly shown that greater learning (or reduced time to learn) is achieved when that learning is spaced out over time. The flow of this course supports learning over a minimum of 5 periods for each "week". Every week there is a 4 part series of homework->lecture->homework->lecture spread over two lecture periods. The last learning period occurs the following week, with a closed lab exercise on the previous week's material. The schedule of topics is shown in Table 1.

Table 1. Course Topics Schedule

Wk	Tuesday	Thursday
1	Introduction	Sequential Execution
2	Static Methods, Parameters	Parameters and Methods
3	Methods, Special Effects	Interactive programs,events
4	Expressions, If statements	If statements
5	If statements, randomness	Counted loops
6	Conditional loops	Midterm
7	Lists/Arrays	Lists/Arrays
8	Excel: Formulas, Functions	Excel: Large Datasets
9	Excel: Explore / Visualize Data with Pivot Tables	Revisit Alice Lists and Searching
10	Material of Student Choice	Alice Project Show

3.2 Course Components

Here we review the various course components, giving examples of and rationale for them.

3.2.1 Exploratory Homework

The goals of the homework are:

- Embed introductory understanding of new topics such that it can be meaningfully tested and deepened in class.
- Develop an appreciation of and skill-set for independent study appropriate to the computing context.

These goals are realized in four (somewhat interwoven) steps:

- Students are introduced to new material, by reading sections of the textbook.
- Students are led through practical activities, either described in the text book or in supplementary notes, to develop their understanding of the reading material.
- Students are encouraged to practice "playing around" with the code they've just developed.
- Between 4 and 6 quiz questions are provided at the end of the homework so that students can self-check their understanding.

The homework description is only a few short paragraphs of text, since most of the instruction is in the book. Crucially, reading is not enough to get a working understanding of the concepts, hence the inclusion of the practical activity in the preparation work.

We view learning to "play around" as an essential part of picking up new languages and systems. We are not referring to unguided "guess and see" tactics for getting code to work; instead, we aim to improve student skills in developing small experiments to deepen their understanding of different concepts. This usually takes the form of questions asking students to make predictions:

- What do you think would happen if you don't specify "penguin" as a parameter where necessary? Try it out.
- If you hit "Play" now do you think anything will happen?

3.2.2 Quiz

Each lecture started with an assessed quiz, acting as an inducement to do the homework: the value of the class sessions is otherwise diminished as unprepared students are unable to engage at an appropriate level during their PI discussions. To further encourage the students to prepare thoroughly, at least one of the quiz questions was drawn from the end-of-homework questions.

3.2.3 Discussion Questions/Lecture

The activities in lecture were designed to engage students in testing and deepening their understanding, while gaining proficiency in effective analysis skills. Clicker questions were developed by going through the day's homework looking for challenging issues, then developing examples where students would select a line of code, predict what code did, or pick a rationale for a provided decision. An example might ask: "The best rationale for why we want to use a function to control a loop is" (for more on PI use in computing see [3,5]).

The process used in class is described in section 3.1 with the modification that one undergraduate tutor was assigned to each group of ~30 students in the lecture halls (engaging them during discussions). This provided more individual attention to students and facilitated effective discussions in our large class. Time spent per question varied, but small group discussions lasted 5-7 minutes each. Sometimes, for challenging concepts, the instructor extended discussion into a mini-lecture or live programming demo. Here, timing is critical. The lecture or demo comes when students are primed to incorporate new knowledge into their existing understanding. Students gained credit for responding, not for correctness. To promote group engagement, students were assigned to groups of three, and their group was required to reach consensus and vote the same answer for the "group" votes.

3.2.4 Labs

Students went to 2-hour closed labs weekly. In introductory programming courses in the US, the work required of students in "closed" lab sessions often represents their first real engagement with new topics. Prior to this, students will usually have only attended lecture or read sections from a textbook. In contrast, students here have already had significant engagement with and feedback on new topics via homework and in-class discussions.

Furthermore, as is typical in skill development, we can expect the students to be progressing at different rates as they work through these developmental exercises – while we expect all students to be able to complete the course; we acknowledge that learning breakthroughs are made at different times for different students.

The Alice labs were designed to address both these aspects of student learning. First, since the students are expected to be well-acquainted with the new topics, the lab is an assessment of their mastery of these topics. Second, students choose one of two options for their lab work. The first option, for more confident students, specifies simply the number and type of Alice constructs that must be used in the development of an Alice world, but leaves the scenario up to the student. The simpler option requires the student to create an Alice world from a more detailed specification. However, after the first couple of weeks, these specifications were more like vague suggestions rather than specific directions (e.g. "one function should check if a fish is "too close" to the pollution).

Excel labs (2) involved data analysis of real world datasets (from the web) and visualization with graphing. The CMU Data and Story Library provided interesting larger datasets amenable to "open-ended" student exploration[1].

[1] Reference thanks to Julian Parris, UCSD.

[2] Adapted by permission from Susan Rodgers, Duke Univ.

"Open" lab hours (where tutors were available for drop in assistance) were offered extensively (both in the departmental lab and directly in the study rooms of the dorms). However, both saw very little use. Anecdotally, students would get help from each other as 60% of them lived in the same set of co-located dorms.

3.2.5 Exams

For the in-class final exam, all but two questions were multiple-choice. They were generated by listing the key concepts in the course, then by developing questions to test understanding via three cognitive skills: code reading and explanation, code writing (select code to complete), and rationale (questions seeking to ascertain appropriate thinking about a concept). The final also included one open-ended question involving code writing (requiring an if statement with compound Boolean expression[2]), and another asking students to analyze a piece of code and "talk me through" what it will do. An out-of-class final was required involving questions on students' perceptions and skills of computing as well as short open-ended reflection questions regarding their role as a student in this course, and their appreciation of computing concepts in everyday software.

3.2.6 Alice Programming Project

For an end of term project students were directed to *Make a digital contribution to communicate your views on an issue facing society. You will do this by creating an animation or game that illustrates, educates and/or prompts viewers to think about the issue and your perspective on the issue.* The project was organized into 3 deliverables, primarily to reduce procrastination. Students had the option to submit to a last-day-of-class "Alice Project of the Stars" competition (voting with clickers). The final deliverable included a reflection where students described what was most challenging about the project and how their experience designing software may have changed their experiences with software, specifically handling software problems.

4. RESULTS

Defining a measurement of results in a general education course of this nature deserves some consideration. Certainly, culturally and logistically, a grade must be assigned reflecting the quality of students' work and their ability to perform certain tasks. In line with many US university courses, 50% of the course grade was derived from performance on exams (midterm and final). However, we did harbor the elusive goal of developing "life-long learning" skills and changed attitudes regarding computing. We were teaching Alice and Excel, but hoped these would impart some of the key computational thinking competencies. This led us to include some assessments where students reflected (openly) on their perception of what value components of the course had been or would be for them in the future. Here we generally report ad-hoc analysis in order to give a feel for the breadth and variation of student experiences. We characterize common themes from the dataset. We seek to show the potential impact of the course, not necessarily the percentage of students with specific experiences.

4.1 Exam Performance

The mean on the in-class final exam was 80% with a standard deviation of 11%. On the code writing question 16% scored perfectly and a total of 40% scored at a level we felt matched a good ability to program (>=6/8). 18% show little ability to write

[2] Adapted by permission from Susan Rodgers, Duke Univ.

code on paper (<=2/8). The average on the code analysis question of a faulty piece of code was much better at 80%. 64% of students scored perfectly which indicated they described the error and how to fix it. 75% of students at least described the error.

4.2 How will computing concepts help you?

One of our first hints that students were getting much more from the course than "just programming" came from a reflection required in the last Alice programming lab. We asked them to tell us the ways in which the computing concepts learned in Alice might help them in the future (acknowledging that knowledge of Alice itself was unlikely to be beneficial). Via an ad-hoc review of answers we found five general areas of interest in students' answers (and we readily admit that a few students answered in uninspired ways: "I could use my knowledge from cse3 to make more creative presentations in a job I might have."). In future work we will perform a more extensive analysis of this data.

- Confidence: *It has given me confidence that I'm able to figure things out on a computer that I never would have thought that I could do.*
- Changed Views of Technology: *Now, every time I find myself playing a video game, I actually understand what makes it work. That these games are not magically produced, that it takes time, skill, and sufficient funds to create these games. I appreciate these games more than before taking this class.*
- Analysis Skills: *Programming allows a person to think more logically, thinking in order and debugging allows the user to gain valuable problem solving skills. Aspiring to go to law school, thinking logically is extremely important and I think this has helped.*
- Communication Skills: *In today's technologically-centered world, using a program like Alice gives us valuable exposure to discussing things technically with other people and explaining clearly what we are trying to do.*
- Organization Skills: *I think that some of these concepts will help me organize my life better. After all, life is a series of choices modeled by if-else statements. If only I do one thing, I gain access to another thing ... These decisions which used to be in the greyish region for me have suddenly become black-and-white.*

4.3 Alice Project and the Impact on Students' General Software Experiences

Programming skills, as evidenced by the Alice project, varied greatly. While the average grade was 85% with a st. dev. of 17%, some students' work was simply superb and others' work showed lack of ability to implement (without direction) basic concepts.

However, driven by the desire to determine the potential long-lasting impact of the experience on students, we asked them to reflect on how they now dealt with software problems in light of their experience as software developers. We asked: *Now that you have designed and developed your own piece of software, how do you react to problems in software you use? How do you feel about encountering those problems? How do you deal with those problems? You may find it helpful to pick a particular experience you have had, and talk about that.*

- Troubleshooting: *This class in general has helped me learn how to troubleshoot a lot more and try to figure out a problem on my own before anything else. Just the other day my older brother was working with excel and he had to count the amount of numbers within different age ranges and he was coming up short by one number every time, so I showed him how to use the filter so he could count it easier.*
- Understanding: *I now understand some of error messages that my software gives me. Such as when it tells me something was set to false. Coincidentally, my laptop completely crashed recently and when I was running a diagnostic check on it, I was able to read some of the error script it gave me, such as the location, duration, and what happened (like if something was set to false).*
- New skills because of increased confidence: *Software is completely a different world from what I've come to know… In a way, I feel like the computational reality makes one feel completely empowered; always in control.*
  Now I try to find the root of my software issues rather than throwing my hands up and determining that it's just a system error that I can't beat. I feel a lot more confident in my interaction with software and can definitely see myself finding more solutions than just "hit restart."
- Seeing concepts in real world*: Now I can think about software [sic] has a logic-based system that will usually work provided that all of its "If"s are satisfied. When I start to think that way, I can properly analyze the conditions that might not be met instead of assuming that someone programmed it wrong.*
- Respect for field: *I now see problems in software to be more expected. After having to build a world and having to go through hours of trial and error, I see how easy it is for software to malfunction or simply not work as expected.*
  I am now slightly more aware of how difficult programming is, and I hope never again to take for granted all the great technology I am surrounded by.

4.4 Impact of Instructional Design

The instructional design of the course was seemingly integral in students' development of analysis and communication skills. However, its tenets and expectations of students' involvement are likely to be a grand departure from the expected student experience. How did they experience this? We asked the following: *Compare and contrast your role in *this* course's lecture with other "standard" university lectures. That is, what are you here to do in this lecture? In other lectures?*

An ad-hoc analysis found students speaking about doing at least one of the following things: participating actively in the course, sharing perspectives/learning/insights with others, studying ahead of time, thinking about and understanding the material, teaching myself/others, listening carefully and focusing, applying knowledge learned earlier, and (a few of) sleeping/being bored.

Some students were quite fantastically impressed with the instructional design and its positive impact and 77% of students would recommend other instructors use this approach in computing courses.

- *Through this course, I was able to learn much more than expected, solely due to the strong emphasis on required participation and gradual development. The constant testing of clicker questions and applied knowledge in the labs requires a steady engagement with the subject, making it much easier to retain the topics learned.*

- *I have not yet encountered a class that encouraged such interest in the subject matter.*
- *At the time I may not have liked this teaching strategy, but now I look back I am thankful because I really did learn.*

5. DISCUSSION

Although the course curriculum of CSE3 had been somewhat pre-defined based on local needs of the university, it was a "good fit" with the stated curriculum/claims of the CS Principles course. This led the instructor to focus on how CSE3 should be defining a general requirement for all (university) students. That is, this would be like "English 101" or "College Algebra" – which in the US are each part of the general educational requirements in almost any university, regardless of student major. From that point of view, the course must focus on a set of long-lasting skills that are likely to transfer or propagate into future computing and technology uses. ***Of the six computational thinking skills defined for CS Principles it is then clear that analyzing problems and artifacts and communicating processes and results are key. The exciting finding through offering the course is that this can (perhaps even must) be done through "teaching programming".*** It is the predictable, simple interface afforded by Alice that is necessary to let students get a handle on and experience the logic of the computer through analysis. It makes clear the level of specificity and detail at which computers work. However, especially when considering the traditional university-style course, it was an instructional design focused on PI that led students to develop those skills. Indeed, students claim these skills are the primary outcome of the course, not an ability to write programs, usually seen as the main outcome of a programming course.

5.1 CSE3 within the Community

In considering the range of courses in the CS community that fall into CS0 or other non-"university majors" courses, we define three variations we've seen – as a starting point for discussing the course reported here:

1. Computing for All: What all humans need to understand regarding the basics of how computers work in order to effectively support the further development of society.
2. Introduction to the Field of Computer Science: A conceptual introduction to the range of areas within computing, as a taster for further study in the field.
3. Preparation in Programming: A course designed to give students the flavor of programming and provide a preparation for a standard CS1 programming course.

From the available literature and our own involvement with this area, these define a range of common learning goals along with a varying target audience (all educated people, or interested career pursuers). In this schema, the original local intention of the course was evenly split between 1 and 3. While we were more focused on core computational thinking skills and access to technology, there was also a need to provide a specific skill (e.g. programming in Alice) that could be leveraged in subsequent (non-computing) courses. Additionally, as a pilot of the CS Principles course, this split across 1 and 3 reflects clearly the 6 computational thinking practices.

However, upon teaching the course (preparation of materials, interaction with students, etc.), the instructor became aware that the ability to program itself was of lesser importance. Partially guided by student reflections, it became evident that the analysis and communication skills they reported gaining really defined how one hopes students would grow in such a course. Having a student make a claim like *now when something goes wrong on the computer, I just work at it and logic it out* is clearly a more satisfying result than *I can make an Alice world that tells a story,* especially for those who may never take another CS course.

The truly surprising and exciting result is that these analysis and communication skills can be developed through the process of "teaching people to program" – as long as it is embedded in a supportive instructional design, as used here. This finding makes a significant contribution to the overall discussion of who should know what about computing and in what order. Certainly it is commonly believed that non-programming activities like those found in CS Unplugged and other similar programmes are good for students as an introduction to computing. However, this experience report suggests that the community discuss the benefits of including programming in curriculum with the scaffolds described here much earlier – with courses supporting course 2 to follow on after that. Additionally, we are led to wonder if this course can actually fully meet the demands of course 1 and 3. We look forward with interest to following the persistence and success of students in this course who go on to take our CS1 course in future terms.

6. CONCLUSIONS

We report on the experience of designing and delivering a pilot AP CS Principles course. The course strives to develop core computational thinking and skills to benefit all 21st century citizens – regardless of profession. From our experience, we posit that, of the six CS Principles computational thinking skills, *analyzing problems and artifacts* and *communicating processes and results* are key. The exciting finding is that this can (and perhaps even must) be done through "teaching programming".

7. ACKNOWLEDGEMENTS

This work was supported by NSF CNS-0938336 and the UK HE Academy – Information and Computer Sciences.

8. REFERENCES

[1] Crouch, C. H., Mazur, E. Peer instruction: Ten years of experience and results. Am. J. of Physics 69 (9), 970–977.

[2] Hake, R.R. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. Am J. of Physics 66 (1), 64-74.

[3] Porter, L., Bailey-Lee, C., Simon, B., Cutts, Q., Zingaro, D. Experience Report: A Multi-classroom Report on the Value of Peer Instruction, submitted for publication.

[4] Roediger, H.L., Karpicke, J.D.. The power of testing memory: Basic research and implications for educational practice. Perspectives on Psychological Science, 1, 181-210.

[5] Simon, B., Kohanfars, M., Lee, J., Tamayo, K., Cutts, Q. Experience report: peer instruction in introductory computing, SIGCSE (2010).