Computing as the 4th "R": A General Education Approach to Computing Education

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

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

ABSTRACT
Computing and computation are increasingly pervading our lives, careers, and societies –a change driving interest in computing education at the secondary level. But what should define a "general education" computing course at this level? That is, what would you want every person to know, assuming they never take another computing course? We identify possible outcomes for such a course through the experience of designing and implementing a general education university course utilizing best-practice pedagogies. Though we nominally taught programming, the design of the course led students to report gaining core, transferable skills and the confidence to employ them in their future. We discuss how various aspects of the course likely contributed to these gains. Finally, we encourage the community to embrace the challenge of teaching general education computing in contrast to and in conjunction with existing curricula designed primarily to interest students in the field.

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

General Terms
Human Factors.

Keywords
general education, peer instruction, active learning, CS0.

1. INTRODUCTION
Computing education is seen as increasingly important, with Wing and others arguing for a grounding in fundamental computational principles for the entire population [24]. Actions are being taken to address this. For example, the UK Royal Society has been commissioned to report on the state of computing education in UK schools [19], and the US National Science Foundation and the College Board are supporting the development an Advanced Placement course, CS Principles, which aims to "broaden participation in computing and computer science."[8]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICER'11, August 8–9, 2011, Providence, Rhode Island, USA.
Copyright 2011 ACM 978-1-4503-0829-8/11/08...$10.00.

In Fall 2010, we ran a pilot of the CS Principles course at a US R1 institution. One of five pilots, ours was unique in that it served the needs of a university general education (GE) course for 570 students. This raised the question of how a GE computing course should be defined, or, put another way, what should every person know, assuming they never take another computing course? In this report, we tell the story of our experiences in putting together a GE course grounded in the CS Principles framework, and of how it impacted students and our views of GE computing.

As we taught the course we paid close attention to the student experience, with all authors attending all lectures and listening-in directly to students' discussions in the class. When prompted (in Week 8), ***the vast majority of students self-reported a range of long-term gains*** as a result of taking the course. Analysis of their open-ended responses shows students reporting increased confidence, changed views of technology in the world around them, increased technical problem solving skills, transfer of computing skills to other areas of their life and increased communication skills. Reports of "learning to program" were very rare (and mostly limited to computing majors). These results were very compelling to us as computing educators, with the following examples being representative:

> *We learned in Alice that computers do exactly what you have them do.*
>
> *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.*
>
> *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.*

We will argue that ***the gains reported by the students form an excellent definition of a general education course in computing***. Notably we believe that engaging students in "learning programming" is critical to the experience – as it provides students very direct control over the computer. Finally, we draw conclusions on the ordering of computing courses at the introductory level to maximize opportunity of access to computational thinking skills development for all students.

2. INSTANTIATING A CS GENERAL EDUCATION COURSE
Here, we briefly overview the instructional design of our course – though we refer the reader to full details in [20]. Our course was based around:

- existing university needs for an academically-rigorous digital literacy course involving logical thinking and the ability to create digital artifacts in subsequent courses,
- the CS Principles framework, particularly the six defined computational thinking practices of analyzing effects of computation, creating computational artifacts, using abstractions and models, analyzing problems and artifacts, communicating processes and results, and working effectively in teams, and
- published experiences in teaching CS0-type courses.

It included 7 weeks of Alice programming [5] and 2 weeks of Microsoft Excel. Alice is a beginners graphical programming environment: graphical both in the sense that programs create and manipulate 3D worlds, and that writing programs consists of snapping visual tiles together on screen. Most critically, we designed the course around a best-practice pedagogy, Peer Instruction (PI) [13], to engage students in deep learning of computing concepts. We made this decision a) based on the evidence from physics and other disciplines that its use dramatically increases learning [7] and b) because it had worked well in previous programming courses [21]. We were particularly interested in PI's ability to focus students on *understanding how programs work,* not just getting them to work.

In the standard PI model, before class, students gain preparatory knowledge typically by reading the textbook and then complete a pre-lecture quiz on the material. In this class, we leveraged the ability to have students work directly in the Alice programming language to assign, exploratory homeworks – which guided them in reading the textbooks for understanding [20]. During class, lecture was largely replaced by a series of multiple-choice questions (MCQs). These typically focused on deep conceptual issues, common student misconceptions or problems [20]. Students followed a process by which they answered a question individually (using a clicker), discussed in an assigned group of 3, and answered a second time. This was followed by a class-wide discussion led by both the students and the instructor. This is the core of the PI pedagogy. Additional course components included a weekly 2-hour closed-lab programming assignment, one midterm, one final, and a multi-week Alice programming project ("make a digital contribution to communicate your views on an issue facing society").

We hoped the PI methodology, with its focus on analysis and discussion, would influence the students' experience positively. Rather than simply "playing around" with Alice, we believed that the PI activities would engage them in the authentic practices [3] that underlie computing experts' thinking and activities; that by asking them to analyze code and discuss it with each other, they would experience via legitimate peripheral participation what actually happens in software developers' cubical walls, or in the IT support center of a major company.

3. STUDENT EXPERIENCE

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. We personally don't believe "ability to write a program" should be a GE course goal. 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.

Though this question may seem leading, the topic had been discussed frequently in lecture, and the question needed to be explicit enough for 500+ students to clearly understand what "kind" of answer we sought. Students were informed that any thoughtful answer would receive full credit. A few responses did imply that students had already "been good" as using computers, and class hadn't changed that. Through analysis of this data, we consider students' perceptions of the "general education in computing" effect of the course.

3.1.1 Methodology

After preliminary, ad-hoc review of the responses (N=521) by two of the authors, one author developed a set of descriptive categories reflecting commonly observed themes. Next that author and one other separately coded a random 10% sampling of the dataset, discussed the results, and refined the categories and descriptions until reaching agreement on that sampling. Then both individually coded a new 10% sampling, reaching an 85% inter-rater reliability (counting matches for agreement on each code for each response). Then one of those authors and the third author coded the remaining data (with the third author reviewing the first 10% sample as a training set).

3.1.2 Results

The categories we identified are shown Table 1, along with prevalence (one response could be coded into more than one category, avg. 2.1), with an example illuminating the category.

4. DISCUSSION

Overall, we were satisfied at the ways in which the students felt the course experiences had impacted them. We patently did not want students to think they were "made to learn programming" and we specifically tried to differentiate the course from one seeking to attract students into the CS major or prepare them to take another programming course. Although the content of our syllabus doesn't differ much from such courses, we utilized the course design to engage students in a different experience - specifically through the in-class peer instruction discussions.

4.1 The Student Responses Define General Education Computing

We argue that the students' statements form a core understanding about what general education in computing should be.

We recognize the students' descriptions as a set of transferable skills and attitudes: confidence to have a go with technology; a new appreciation/awareness of that technology; problem solving skills to plan out solutions to problems and then to enact them, detecting and correcting bugs along the way; and communication skills appropriate for discussing issues about computing systems.

Table 1. How The Course Will Help in Your Future: Categories of Student Responses ordered by Prevalence.

Category	Example Response
Transfer, Near (64%): can apply new skills in software use	*Using new machinery like sound editing equipment … will require the ability to manipulate and design using the basic commands to form unique creation. Similar to Alice we will be restricted to the amount of actions we can perform sometimes but through our creativity we can manipulate the basic commands of the music program to create variations not standard to the system. Like how we mad[e] frogs appear to be hopping when in actuality the Alice program does not have a specific method that makes frogs hop.*
Personal Problem Solving Ability: Debugging (39%): can logic it out, attempt to or deal with unexpected behavior	*I have learned how to target problems when I am working on a computer and use the process of elimination to try to fix the problem instead of just restarting the computer like I used to. This skill partially developed from taking CSE3 and becoming more comfortable with working with new computer programs and dealing with bugs in Alice.*
Personal Problem Solving Ability: Problem Design (29%): can develop plan to solve technical problem, can see what requirements exist	*We learned in Alice that computers do exactly what you have them do. Using this knowledge, we can understand how programs like Excel and Numbers work and learn that when we are using these programs, we need to specify and be exact with what we are doing in order for the programs to meet our needs and plans.*
View of Technology (25%): greater appreciation or understand 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.*
Transfer, Far (23%): can use problem solving skills in other areas of life	*I feel that learning the language of computing definitely helps you understand dense reading a lot more efficiently. I personally have noticed that my in-depth understanding of Computer Science wording has helped me understand my mathematical theorems and proofs more regularly than before.*
Confidence (21%): increased ability to do things on computer, a can-do attitude	*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.*
Communication (7%): communicate better about technology	*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.*

To rate the value of these skills, consider the typical knowledgeable IT person, the colleague any office worker calls over when they're having trouble with their PC. He or she is the confident problem solver who can talk to you about your problem. Even though they may not know directly about your software or your issue, they know they'll get there with some educated exploration. Their skills and attitudes bear a striking similarity to those described by our students.

As to whether such skills should form part of a general education requirement, there are two pertinent questions: do all citizens need this skill/attitude set; and is it necessary to formally teach it? The recent push for a broader computing education indicates that society is beginning to accept the importance of computing skills for all; and we use Turkle [23] to argue that a concerted effort is required. Turkle argues that the adoption of computing technology to support our thinking processes has in fact shaped the way we think. Specifically, the Apple Macintosh-style direct-manipulation interfaces introduced in the 1980s encourage us not to look under the surface and not to attempt to understand or appreciate systems deeply. She argues that we have been seduced into an expectation that systems will be easy to use and we are surprised and unprepared when they aren't.

As an example, consider a modern word processing package – a far cry from early, glorified text editors like MacWrite. The underlying document model of the modern version would have been the domain of a professional typesetter in years gone by, yet users expect to be able to intuit the model largely via direct manipulation with what they see on screen.

The combination of increasing complexity with incorrect expectations only leads to frustration. When software does something unexpected, most users have no training in how to go about understanding what is going on, and few skills in identifying or correcting the problems they are experiencing. Consequently, to them, software has become something magical and beyond their control.

We can relate each of our students' response categories to the manner in which Turkle's argument suggests most computer users are likely to think.

- *Confidence*: Software systems are too complex for me to understand. When they don't do what I want, I don't know what to do. I can't have an effect.
- *Appreciation*: I don't have any insight into how the technology works and I've never been encouraged to look "under the hood".
- *Problem solving*: Software and computers are meant to be easy to use – I shouldn't need to plan ahead to complete my task; when something happens I don't expect, I haven't a clue where to start – I have to get someone to help me.
- *Transfer*: I've only just mastered Word. Now I've got to start all over again with Excel. Nightmare! It's a different world.
- *Communication*: I can't get the IT person to understand my problem at all. It's as if he's from a different planet.

We suggest that through our GE course, students gain the ability to balance the inherent complexity of software against the knowledge that, with effort and use of appropriate skills, they can

understand the software or "figure it out". In particular, they can understand the complex models underlying software via a process of inductive reasoning based on experimenting with the software.

4.2 Comparison with Existing "First" Computing Courses

In schools, there exist current courses that could possibly be viewed as a GE in computing, varying from training in the use of IT, through programming courses, to the introduction of computer science concepts. We assess whether these styles of courses are likely to deliver experiences that our students described.

Before beginning, we acknowledge Papert's early radical general intellectual training based around programming in Logo [16]. There is much commonality between the skills he describes his students developing and those described by our students. A key difference is that of scale – our students are in a traditional mass education system whereas Papert describes a more personalized self-exploratory learning environment.

4.2.1 IT training courses.

IT training is typically centered on the direct use of typical office-oriented packages. For example, the Scottish education system features a 5-14 Information and Communication Technologies (ICT) strand in its national curriculum – traditionally involving follow-the-steps-style worksheets [11]. Assessment often features simple factual recall or production of artifacts – and transferability of skills is hard to assess. Crucially, such courses drive towards outcomes such as "I can create a PowerPoint presentation", rather than anything to do with the understanding of or communication about how to be an effective IT user. In a survey of over 2000 Scottish school pupils [14], it was clear that this curriculum was found to be both boring and a totally inappropriate forerunner to later computing courses. Worse, anecdotal evidence suggests that many incoming university students are barely-adequate IT users. Furthermore, contrary to popular opinion, Bennett [2] demonstrates that the evidence for Digital Natives [17] is far weaker than is widely reported.

4.2.2 Preparation for programming courses

These courses introduce the excitement of creating programmed artifacts without going into the traditional heavyweight detail of a standard CS1. Examples include courses that use robots or the Scratch[18], Alice, or Greenfoot [10] programming environments.

We are unable to ascertain whether students taking these classes have also experienced changes similar to those our students report – though published work does not report such findings. In [15], students' attitudes regarding interest in computing increases in an elective Alice-based CS0 course. Our students were given the same survey, but no statistically significant increase in attitudes occurred – perhaps because students' interpretation of the terms in the questions changed from pre-test to post-test; perhaps because they did not choose to take the course and were not as likely to be pre-disposed to come to like computing. In future work, we seek to better understand this result.

We speculate that the focus in these courses is typically on the excitement of getting programs working, rather than on the deep understanding and articulation of what the students did. For example, in [18], the digital fluency associated with Scratch involves "designing, creating and inventing". Teachers of course do want the deep understanding, but much of the student activity and assessment, where there is any, is most-likely focused on "can you do it?" As Section 4.3 shows, we view the core difference between our course and other programming-oriented courses is the emphasis on articulating deep understanding.

4.2.3 Non-programming introduction to computer science: Excite programs

There is a wide range of programs that aim to introduce computer science without involving programming at machines. The most well-known of these is CS Unplugged [1], and author Cutts has run a similar effort called CS Inside [9]. Both the US and the UK are considering adopting aspects of these programs into nascent school curricula. We refer to these as *excite* programs, because a key aim is to excite participants about core aspects of CS in order to increase enrollment in future computing courses. Indeed, the origins of both programs lie firmly in university outreach activities. The activities of the programs were originally designed for one-off, non-assessed sessions where excitement is the core goal, with learning as a secondary goal. They do use active and often kinaesthetic learning methods that undoubtedly are highly engaging for the participants.

We speculate that the learning activities of these programs will not form an effective general education, as our students' responses define it, for a number of reasons:

- Their main focus is to raise awareness of a broad range of computer science topics, (e.g., data representation, algorithms, cryptography, intractability, etc.) rather than on a narrower core set of transferable skills and attitudes.
- Whilst the active learning embedded in the activities does foster core skills such as problem solving and group work, or core attitudes such as the deterministic nature of algorithms (and hence programs and computers), the rather self-contained nature of each learning activity goes against on-going step-wise development of these skills.
- Their separation from the world of software and machines is likely to make transfer of core generic realizations about the structure and use of computer systems difficult.

4.2.4 A matter of speculation

Here, we have only been able to speculate that alternative course formats considered for introductory computing do not effectively fulfill a general education role. We urge those teaching any of the formats covered here to replicate our open-ended reflection question, presented in Section 3, with their students. Particularly interesting would be the effect on students taking such courses as a *requirement*, as ours did, and not by elective choice.

4.3 Key Effects of the Instructional Design

The Peer Instruction Effect. We believe our instructional design centered in analyzing code (in homeworks, discussion questions in class, and (naturally) programming labs) impacted students. Certainly, instructors hope students in programming courses with standard lecture develop code analysis skills, but it is rare that we focus class time engaging students in that practice for themselves. Even in lab-based lecture environments, students' work with live programming may not engage them in analysis. As Stephen Cooper advised us [4], some students may just play around randomly trying things until they get the desired result. From our classroom observations (two authors observed and engaged students in their group discussions during lectures), the use of PI gave students the opportunity to viscerally develop the understanding that computers are, likely contrary to their previous experiences, deterministic, precise, and comprehensible. Through vigorous, constant engagement in the struggle to not just *create* programs or *learn to use* computing concepts like looping and

abstraction, but instead to *analyze, debug, and critique* Alice code, students seem to have internalized these three core attributes of computational systems. We see evidence of this in some students' responses regarding their experiences when something goes wrong on the computer. They now recognize the problem might be the fault of the computer or it might the fault of the user. This stands in contrast to their stated previous beliefs that it was always their fault (or in some cases always the computer's fault). This seems a critical first step in an increased sense of empowerment that should serve them positively in their futures.

Furthermore, the general education literature provides strong evidence in support of the PI process as a way of promoting deep learning. Teasley [22] demonstrates that speaking out one's understanding improves learning; articulating it to a peer even more so. Craig et al. [6] show that paired learners gain as much from watching a video of a tutor at work with another student as from one-to-one tutoring – interesting for the similarities to class-wide discussion (a form of dialogue between individuals seeking clarification and the instructor). Finally, Karpicke has shown in a number of studies, e.g. [12], that testing promotes more learning than studying. We are testing students in every class session, both with the quiz and discussion questions.

Programming with a Visual Execution Model. Could we provide students an equivalent experience by teaching a PI-based course using Excel or other computing applications? Our experience suggests the value of a visual, scaffolded novice programming environment like Alice is that it provides students the most direct form of interaction with the computer possible – programming-language-level control without the distraction of syntax errors and in a way such that every part of their program's execution is visible to them (we didn't cover the topic of variables). Crucially, the mapping from their program code to an observable execution model is very straightforward. To the extent that other existing or future environments meet these criteria, we believe they would work effectively, too. Key is that students engage with a basic programming interface that manages cognitive load, enabling them to focus on core computational concepts.

Instructor Recommendations. Specifically because the technical content of this course matches that of typical introductory programming courses, it is especially important for the instructor to stay focused on the GE goals of the course. It is challenging to change one's habits from rewarding and assessing success in *creating* programs to success in *analyzing and communicating about* programs. How does this challenge play out in class? While clicker questions in class may ask students to select a line of code to complete a program, or to read a program and select a description of what the code does – the manner in which the instructor must interpret students' clicker votes to the question must reflect the goal of analysis, not correctness. Even if more than 95% of the class gets a question correct, that doesn't mean that students have a thoroughly correct understanding of why the answer is right. Moreover, they must still be given the opportunity to practice discussion of the question, providing their explanations to each other, engaging in interactive questioning and justification, and modeling for each other methods of thinking about the problem. In class-wide discussions, as many students as possible should be asked to explain in their own words, both why the correct answer is correct, but also how they figured out the other answers were wrong.

Even more challenging for the instructor is to consider completely different kinds of questions than one traditionally asks on introductory programming exams; questions that ask what is the best explanation of why something is (e.g. why do we used a counted (for) loop instead of a while loop) and even questions (on exams) that ask students to not only give an answer, but to explain their analysis that led them to that answer. Testing whether students can merely "write code", with no other explanation or analysis required, seems to be of limited importance.

4.4 General Education First: An Issue of Equity?

From our experiences of deep reading of students' reports on the impact of the class, we propose that one feature underlies many of our coded categories: the experience of coming to a *new understanding of what a computer is and how one can interact with it*. Overall students seem to grasp 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.

Is it possible that this visceral understanding (compared to acceptance of telling or quasi-belief) lies at the core of the development of computational thinking skills? Moreover, if one does not yet have this core understanding (as it seems many of our highly-selected college students did not), what is the impact of, for example, a CS Unplugged activity on cryptography, or a course on using Excel effectively for data analysis?

Author Cutts has extensive experience of working with Scottish school teachers and pupils to instill discipline-appreciation through CS Unplugged-style activities. From his experiences, students may overwhelmingly report increased excitement or interest from these experiences, but measurements of learning vary – with a large portion of students seeming having missed even the basic points of the session. This is reflective of learning reports in introductory computing courses. Even in those courses (perhaps CS0) targeted to work with students of any ability, the performance gap for some students seems unassailable. Every instructor has anecdotes of students trying earnestly to master programming, but still failing, if not the course, then failing to develop deep understanding of the core concepts. It is only natural, given repeated experiences, that this may lead instructors to adopt a fixed mindset regarding *some* students' abilities to program. The myth of the programming gene is not so easily dismissed by any experienced instructor.

We posit that lack of understanding that computers are deterministic, precise and comprehensible may be a key factor leading many to struggle, seemingly in vain. Certainly, many students might enter our courses lacking this belief. But some may come to develop it on their own and others may simply be willing to accept yet more incomprehensible magic in the process of programming. We suggest that only some students, with a possibly indefinable set of life experiences, enter our classrooms believing computers can make sense and be reasoned with. Reiterating Turkle's argument [23], as computing has embraced "more intuitive" human interfaces, we have likely actively discouraged attempts to reason about computer interactions.

Core Competencies Before Appreciation. We propose that the community further study the effect of combinations of general education and *excite* or discipline-appreciation courses. Based on our students' claims of the confidence and ability they will have in future engagement with computers and in their increased understanding of where computing concepts exist in their everyday technology use, we propose excite and discipline-appreciation courses will be much more effective when preceded

by a GE computing course. As a comparison, multiplication (let alone any advanced mathematical concept) is likely a mystery when taught to students lacking understanding of addition.

It's true, as outreach instructors, we may not have as much fun or personal excitement in teaching a course with the design and goals as outlined here. Not surprisingly, English teachers usually prefer to teach specializations such as poetry or Shakespearean Literature over basic composition. This may be a combination of the fact that students have already moved a bit up the expertise ladder making them easier to communicate and work with. It may be because these courses allow an instructor to better share their passion for a deeper and more nuanced engagement with their subject. It may be that students are more likely to be in such courses based on their own choice, rather than as a requirement. But we suggest that instructors consider the deeply rewarding contribution that lies in opening the eyes of all to the skills and attitudes required to live in the computing age.

Where Have You Left Them? Is 7 weeks of Alice and 2 weeks of Excel, with a carefully supporting instructional design, sufficient to define the grounding in the fundamental principles of computation? Perhaps not. This course didn't even cover variables. Yet students seem to feel they have been given the keys to do something useful, something meaningful – with a minimum subset of computational elements. Given more time, one can prioritize more experiences or understandings we want all citizens to have. However, unless *starting* with programming, these efforts will be hamstrung. We look with interest to see how others adopt and expand this curriculum. Interestingly, by the end of this course, students not only change their views on computing, but they get a significant springboard into traditional introductory programming education. In the short term, this seems a valuable component of any computing course taken by many.

5. Conclusions

We encourage the community to consider the needs of a GE curriculum in computing – in contrast to and in conjunction with courses designed to interest students in the field. We provide an example of engaging best-practice pedagogy in teaching a supportive programming language (e.g. Alice) and see that students report gaining long-term skills and confidence as a result of the course, outcomes that we view as core for a GE in computing. Based on our experiences, we hypothesize that GE computing courses should be taken before other computing courses: including application skills courses, excite courses, or more mainstream programming courses. Moreover, we posit that doing so is a key matter of improving the equity of access to learning in those courses. We encourage the computing education community to engage with GE courses that lift the veil of secrecy and elitism from the field and use of computing.

6. ACKNOWLEDGMENTS

This work was supported by the NSF CNS-0938336 and UK's HEA–ICS. The authors thank Sally Fincher and Steve Draper.

7. REFERENCES

[1] Bell, T., Alexander, J., Freeman, I. and Grimley, M. 2009. Computer Science Unplugged: School Students Doing Real Computing Without Computers. *New Zealand J of Applied Computing and Information Technology*, 13(1), 20-29.

[2] Bennett, S., Maton, K. and Kervin, L. 2008. The 'Digital Natives' Debate: A Critical Review of the Evidence. *British J. Educational Technology*, 39(5), 775-786.

[3] Brown, J.S., Collins, A. and Duguid, P. 1989. Situated Cognition and the Culture of Learning. *Educational Researcher*, 18(1) 32-42.

[4] Cooper, S. 2010. Personal communication.

[5] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *J. Computing Sciences in Colleges*. 15(5) 107-116.

[6] Craig, S., Chi, M. and VanLehn, K. 2009. *J. Educational Psychology*, 101(4), 779-789.

[7] Crouch, C. and Mazur, E. 2001. Peer Instruction: Ten years of experience and results. *Am. J. Physics. 69 (9) 970-977.*

[8] CS Principles website: http://csprinciples.org

[9] Cutts, Q., Brown, M., Kemp, L. and Matheson, C. 2007. Enthusing and informing potential computer science students and their teachers. *SIGCSE Bulletin* 39(3),196-200.

[10] Henriksen, P. and Kolling, M. 2004. Greenfoot: Combining Object Visualisation with Interaction. *Companion to 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 73-82.

[11] Information and Communications Technology: 5-14 National Guidelines. 2000. Learning and Teaching Scotland.

[12] Karpicke, J. and Blunt, J. 2011. Retrieval Practice Produces More Learning than Elaborative Studying with Concept Mapping. *Science* Vol 331, no. 6018, 772-775.

[13] Mazur, E. 1997. *Peer Instruction: A User's Manual*. Prentice Hall, Saddle River, NJ.

[14] Mitchell, A, Purchase, H.C. and Hamer, J. 2009. Computing Science: What do Pupils Think? *14th ITiCSE*, 353.

[15] Moskal, B., Lurie, D. and Cooper, S. 2004. Evaluating the Effectiveness of a New Instructional Approach. *35th SIGCSE,* 75-79.

[16] Papert, S. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, New York.

[17] Prenksy, M. 2001. Digital natives, digital immigrants. *On the Horizon*, 9(5), 1–6.

[18] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E. Silver, J., Silverman, B. and Kafai, Y. 2009. Scratch: Programming for All. *Comm. ACM* 52(11), 60-67.

[19] Royal Society. 2010. *Current ICT and Computer Science in Schools – Damaging to UK's Future Economic Prospects?* Press release.

[20] Simon, B., Esper, S. and Cutts, Q. 2011. *Experience Report: an AP CS Principles University Pilot*. Technical Report CS2011-0965. University of California at San Diego.

[21] Simon, B., Kohanfars, M., Lee, J, Tamayo, K., Cutts, Q. 2009. Experience report: Peer instruction in introductory computing. *41st SIGCSE*, 341-345.

[22] Teasley, S. 1997. Talking About Reasoning: How Important is the Peer in Peer Collaboration? In *Discourse, Tools, and Reasoning: Essays on Situated Cognition*. Springer-Verlag, Berlin, 361-384.

[23] Turkle, S. 2003. From Powerful Ideas to PowerPoint. *J. Research into New Media Technologies*, 9(2), 19-28.

[24] Wing, J. 2006. Computational Thinking. *Comm. ACM* 49(3), 33-35.