SIGCSE 2014 Trip Report - Chris Wilcox

3/22/2014

INTRODUCTION

Two weeks ago I attended the Special Interest Group on Computer Science Education (SIGCSE), held from 3/6-3/8 in Atlanta. Over 1300 faculty, instructors, and secondary teachers attended the conference. Many attendees are teaching faculty or are doing active research in Computer Science education, or both. The conference continues to be valuable, especially for instructors that teach introductory courses. In the first two pages I summarize some of the key ideas from the SIGCSE community, and explain how I think they can contribute to what we are doing in our own introductory courses. In the remaining sections I present some details on the 12 technical papers, 2 special sessions, 1 supporter session, and 2 keynote speeches that I attended. Finally I talk about the research I hope to pursue. Please send any comments to wilcox@cs.colostate.edu.

EXECUTIVE OVERVIEW

This section summarizes the practices that are being discussed at SIGCSE. The emphasis on the Inverted Classroom, Peer Instruction, Pair Programming and Media Computation continues, and many attendees seem to consider these as entrenched best practices. In addition there was research presented on incorporating MOOC Techniques into the classroom, and many references to using the Object-Early, Classes-Late approach in introductory programming courses. I also made an effort to understand how other institutions incorporate Discrete Math into their curriculum. Below I cover the techniques that are current in classroom pedagogy, including the new topics mentioned above, and a recap of the techniques I discussed in my report from last year with minor updates.

OBJECTS-EARLY, CLASSES-LATE: Based on the sessions I attended, this is the most common model for teaching introductory programming courses at other institutions. Instead of trying to teach procedural programming, **object-early** recognizes that languages such as Java and Python require early knowledge of object-oriented principles. This is something we already do, as we have found it very difficult to avoid classes and objects completely. For example, the use of String and Scanner inherently require the student to instantiate these objects and call their methods; so why not present the object-oriented methodology from the outset? The term **classes-late** means that the details of classes (public vs. private, static keyword, inheritance, polymorphism, overloading, overriding, reflection) are deferred until later in the curriculum, which seems to me like a good idea.

MOOC TECHNIQUES: A new trend this year is the use of MOOC technology for teaching on-campus, and a couple of studies compared the effectiveness of this approach versus traditional lectures. The claim was made that this approach is equally effective to normal classroom teaching, but I would note that many more resources were applied in the classroom form of a MOOC, including programming labs, help desks, and other teaching assets. The authors of the MOOC papers warn that the creation of materials for online courses, such as teaching videos and automated exercises, requires considerable upfront resources.

DISCRETE MATH: I investigated how other institutions handle discrete math. In the interest of full disclosure, my opinion is that the integration of math into our introductory course with a high number of non-majors is less than optimal. My point of view is now bolstered by considerable research. I found that most other schools have more separation between major and non-major courses, allowing them to customize the content for each audience. A number of schools offer an introductory programming course that blends majors and non-majors, but these

courses are often tailored to attract under-represented students and/or help students with weaker backgrounds make an entry into Computer Science. I was unable to locate any schools that integrate discrete math into their introductory programming course; instead the predominant model is to offer it as a standalone course at the sophomore level. What makes us unusual (if not unique) is that we integrate discrete math into an introductory course that has more non-majors than majors. To get this information I talked with faculty and/or browsed the posted undergraduate curriculum for a number of universities, including top schools (Stanford, Berkeley, Cornell), teaching centers (Universities of Toronto, Michigan, Washington, San Diego) and nearby institutions (Universities of Utah, Colorado, Montana). I have not yet identified another institution that uses our approach.

The remaining techniques were discussed in my report last year. Many claims continue to be made about improvements in student participation and performance, and retention of under-represented students based on these practices. The most impressive statement was from Berkeley, where the Joy of Computing program has succeeded in attracting more than 50% female participation in introductory Computer Science.

INVERTED CLASSROOM: The inverted classroom assigns textbook and slide reading in advance of classroom time, enforced by online assessments. Instead of lectures, classroom time is used for interactive activities that engage students, for example, peer instruction or in-class programming activities led by the instructor.

PEER INSTRUCTION: This is the activity that most commonly replaces traditional lectures, i.e. slide presentations. One version of peer instruction involves teams of students using clickers to answer questions designed to stimulate discussion. There appears to be a high rate of adoption of different kinds of peer instruction.

MEDIA COMPUTATION: This pertains to the assignment of programming exercises that incorporate graphics, video, and audio in place of having students write console programs that input and output text. It's hard to imagine a downside to providing stimulating exercises, other than the extra work it entails.

PAIR PROGRAMMING: Instead of discouraging collaboration as we do, students are paired with other students for programming activities, including lab work and programming assignments done outside the lab and classroom. The claim is made that this is a highly productive and popular activity in terms of student engagement, but I also have heard the concern that weaker students may not learn as much because the stronger students do all the work.

PROPOSED PRACTICES

As far as the application of the techniques above, I feel that our current use of Automated Testing, Media Comp, and Peer Instruction has been successful and should be continued. At some point in the future I would like to conduct experiments in Pair Programming and Code Inspection in the controlled environment of recitations to see how these work out. In addition I recommend that we continue recruiting undergraduate assistants of both genders to handle much of the recitation teaching, as this has been very successful. I do not see any need to incorporate MOOC techniques or to push the Inverted Classroom any further than our current approach, in which we offer 2 lectures, 2 recitations, and 1 peer instruction session weekly. The division of the recitation into two separate sessions has some problems, but I continue to believe that the benefits outweigh the disadvantages. I will address the incorporation of the Object-early, Classes-late methodology and changes to where Discrete Math is taught in a separate letter to the undergraduate curriculum committee.

MOTIVATION

This section is repeated from last year. The Computer Science education community has a mound of research related to the efficacy of the various pedagogical practices shown above. A handful of basic motivations link the disparate techniques:

ENGAGEMENT to overcome OBSTACLES: The research says that students who are engaged learn more, that is students that are actively participating in constructing solutions are more likely to learn than those passively listening to a lecture. Peer Instruction is one of the primary tools used to increase engagement. The theory is that early computer science makes students face cognitive hurdles that are hard to overcome. This makes sense to me after seeing students trying to cope with the Linux operating system, Eclipse development tool, Java programming language, and our own website and process, all within the two thirds of the semester we spend on programming related activities! Techniques such as Media Computation increase the level of satisfaction when obstacles are overcome, and the claim is made that Pair Programming allows students to encourage each other to continue when difficulties are encountered.

COMMUNITY and COLLABORATION: The research says that students who belong to a community learn more, especially for groups that traditionally underrepresented in Computer Science. Research shows that students that feel a sense of community in their classes or departments are much less likely to leave Computer Science. Peer Instruction and Pair Programming both create community, and some studies have shown that they increase retention and decrease the number of dropouts. I should also mention that mentoring is a big part of creating community; many institutions make the assignment of mentors a priority, dedicating significant resources to make sure students are not left on their own.

KEYNOTE SPEECHES

The first keynote address was *Computational Thinking for All* by Robert Panoff from the Shodor Education Foundation. This talk was entertaining, but I'm not sure how much substance there was. The basic premise was that we spend lots of time "interacting with computing and communication tools", but somehow we are not addressing the things that really matter, which in the opinion of the presenter are "quantitative reasoning, computational thinking, and multi-scale modeling". An educational framework that allows students to develop these concepts was presented, and looked fairly interesting.

The second keynote address was *Transforming United States Education with Computer Science* by Hadi Partovi. The speaker is the founder and head of **code.org**, and his talk was extremely interesting. The pillars of code.org are to 1) *Educate* by bringing computer science education to K-12 schools, 2) *Advocate* by removing legislative barriers, and 3) *Celebrate* by countering negative stereotypes of Computer Science. His organization sponsored the "Hour of Code" which had amazing participation levels: 28 million students and 35 thousand schools in 170 countries, with 48% female participation and 97% of teachers reporting a positive experience. Another activity of code.org is to develop curriculum for a 20-hour introductory course for K-8 students. They are also partnering with school districts to provide a professional development workshop to K-12 teachers. Sponsors include Google, Microsoft, NSF, Yahoo, Amazon, and many other companies. I showed the code.org videos on why you should learn computer programming from Barack Obama, Bill Gates, Steve Jobs, and Mark Zuckerberg to my class, you should definitely take a look at these if you have not already!

SUPPORTER SESSION

Google, Microsoft, and other companies that sponsor SIGCSE are allowed to provide supporter sessions. Normally I avoid this kind of advertising, but I decided to attend the GitHub training session after talking with an employee from their education team. The session was called *Version Control with GitHub: A Foundation of Collaboration.* The company has an interesting revenue model – online repositories can be created and maintained for free for academic and open source projects – thus their revenue is based only on commercial users. GitHub has made their support for educational institutions extremely attractive, and I am considering substituting it in place of Subversion in the courses I teach that use source control. My reason for this is that GitHub appears to be quickly becoming the *de facto* leader in source control, with extensive use in open-source and academic projects. It also has arguably the best cross-platform support in the form of both command-line and GUI applications for Windows, Mac, and Linux. The session was essentially a very well run tutorial in which the basic model for GitHub was shown along with lots of practical commands and usage.

SPECIAL SESSIONS

The first special session I attended was the report on the IEEE/ACM Computer Science Curricula 2013, which has recently been released. Robert France was on the committee and can no doubt give a better summary than me. I was impressed at the effort that has been made towards establishing a comprehensive curriculum in each area of Computer Science. I have a flash drive with the entire report that I will make available on one of the servers.

The second special session I attended was called "Rediscovering the Passion, Beauty, Joy, and Awe: Making Computing Fun Again". This movement is apparently based on comments from Grady Booch at a previous SIGCSE. This is the session where Daniel Garcia from Berkeley presented their astounding rates of female participation, at least partly based on the "Joy of Computing" curriculum they have developed. We also heard from Jennifer Campbell at the University of Toronto, another hotbed of CSE research, who referenced the work of Beth Simon and discussed many of the best practices described above.

BIRDS OF A FEATHER

I attended a session called *Using and Sharing Programming Exercises to Improve Introductory Courses* by David Hovemeyer (York College) and Jaime Spacco (Knox College). Jaime was the main author of the *Marmoset* automated grading system. Several resources for finding already prepared assignments were mentioned, including *Ensemble*, *Problets*, *CodingBat* and *Cloud Coder*. The latter is the new tool being worked on by Jaime. An interesting idea came up in this session, which was a standardized format for assignments that would allow exchange between different institutions. This year there was unfortunately not a session on automated grading.

TECHNICAL SESSIONS

I attended four technical sessions, each with three papers. These included *Automated Assessment*, *Assessment and Evaluation*, *Peer Instruction*, and *What we Say, What They Do*. Both of the assessment sessions and the peer instruction papers were of great interest to me. Here are the papers for these sessions, listed by title and the primary author, along with a synopsis of what was presented:

AUTOMATED ASSESSMENT

Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units, Nickolas Falkner (University of Adelaide). In this paper the authors made use of a proprietary automated testing

system to compare the number and quality of student submissions based on the granularity of the feedback provided. The paper is interesting for us because we have limited granularity based on the split between preliminary and final testing that we provide in our automated test system.

Adaptively Identifying Non-terminating Code When Testing Student Programs, Stephen Edwards (Virginia Tech). This was a paper that is highly relevant to our automated testing system, which like the authors' system needs to handle student programs that do not terminate. The authors describe a system for detecting and handling such programs using adaptive timeouts. One of the more interesting aspects of this paper is that the authors don't seem to differentiate between types of non-terminating programs, for example infinite recursion versus infinite loop versus deadlocks based on an incorrect number of inputs. This is a possible topic for a future paper.

Can Computers Compare Student Code Solutions as Well as Teachers?, Matheus Gaudencio, *(*Universidade Federal de Campina, Brazil). This paper investigates issues related to automatic grading and plagiarism detection. Their approach was to evaluate a number of different automated strategies for code analysis of student programs. They compared these techniques to actual instructor evaluations. If I understood their results, it appears they found that the computer was just as good at detecting code differences, plagiarism, as well as identifying problems with student approaches.

ASSESSMENT AND EVALUATION

Importance of Early Performance in CS1: Two Conflicting Assessment Stories, Leo Porter (Skidmore College) and Daniel Zingaro (University of Toronto). This was one of the first studies I have seen on early performance problems. Apparently previous work has identified three reasons for the bimodal distribution of performance that many instructors, including us, have observed. The first explanation is *Geek Genes*, which explains superior performance as a combination of innate talent and/or help from parents that are engineers or computer scientists. The second is *Stumbling Block*, which postulates that early misunderstandings can cause severe performance problems at a later time. The third is *Learning Momentum*, which says that early success leads to future success and early failure leads to future failure. The conclusion of this research is that early performance is indeed highly correlated to later success, making it crucial to intervene early if you want to change outcomes.

Reinventing homework as cooperative, formative assessment, Don Blaheta (Longwood University). The research in this paper proposed a very specific teaching technique for non-introductory classes that require algorithmic thinking. The authors implement a model where the students work cooperatively to make an initial code submission and are given detailed feedback. This is followed by a final submission that is graded without feedback. The conclusion of the work is that early written feedback is more valuable than grading, and that cooperative work is better than individual work, but this leaves out many details of how they evaluate group work.

Evaluating an Inverted CS1, Jennifer Campbell et al. (University of Toronto). This was a comparative study in which they split a very large CS1 course into multiple sections with and without an inverted classroom. In the inverted classroom they used videos and other materials from a MOOC, in the other classroom they employed the traditional lecture format that had previously been used. Both instructors believed their way of teaching was the best. The bottom line of this research was that the different sections had similar drop rates and exam scores. A couple of interesting observations were made, including much lower attendance in the inverted classroom. The study was performed on a large class (600+ students) in which 76% of students were not intending to continue in Computer Science, and 62% of students had no prior programming experience. The aim of this research seemed to be to evaluate the effectiveness of lectures.

PEER INSTRUCTION

Peer Instruction Contributes to Self-Efficacy in CS1, Daniel Zingaro (University of Toronto). In this paper the author studied whether peer instruction helped increase both self-efficacy and exam graded in a CS1 course. Self-efficacy has to do with the self-confidence shown by the student in their ability to program and solve problems. The outcome was that peer instruction clearly contributes to self-efficacy and there is some evidence that it also improves test scores.

New CS1 Pedagogies and Curriculum, the Same Success Factors?. Christine Alvarado (Univ. of California San Diego). This paper studies whether new pedagogic techniques such as Media Comp, Pair Programming, and Peer Instruction change the influence of traditional success factors. These include prior experience and the amount of confidence that students have in themselves. The very interesting conclusion of this paper is that prior experience remains a large factor for male students, but not for female students, and that confidence problems can be exacerbated by group activities. However, the author claims that in general the new pedagogic techniques decrease fail rates, increase retention, and help bridge the gender gap commonly found Computer Science.

Social Effects of Pair Programming Mitigate Impact of Bounded Rationality, Zhen Li and Eileen Kraemer (University of Georgia). This research compared pair programming to solo programming. The authors found that pair programming resulted in the earlier application of critical thinking, i.e. pairs identified problem areas much more quickly than individuals. However, results on the programming quiz used as the basis for the study were identical.

EXHIBITION

At the exhibition I made an effort to visit vendors with automated assessment or content tools. The main player in this area is Web-CAT, which seems to be the most widely adopted automated testing tool. The tool appears to be excellent, with many useful features, however it is primarily targeted at Java Programming and promotes the use of Test-Driven Development by requiring students to develop JUnit tests along with their code. Web-CAT appears to be the only standalone testing framework that is not tied to content. Another class of online tools has fixed content created by the provider. These include Turing Labs, the provider of the MyProgrammingLab tool we previously evaluated, and Zyante, a new player in the market with a very interesting offering. I signed up for access to an evaluation version and plan to evaluate it.

RESEARCH OPPORTUNITIES

I identified a couple of opportunities for research papers that I plan to pursue this summer. The first paper I am proposing is *An Evaluation of Automation in an Introductory Computer Science Course* in which I plan to discuss the various tools we are using for Automated Grading, Peer Instruction, and Online Tutorials and Assessment. The key topic here is a quantification of the decrease in teaching resources and of course the effect on student performance. From the data I have gathered so far, I think we can make a good case that our systems are improving student attendance, engagement, and performance, while allowing us to handle increasing enrollments without a proportionate increase in teaching resources. The second paper I am proposing is *Modified Regression Testing Techniques to Support the Automatic Grading of Student Programs*. This paper would explore the different methodologies I have developed for grading student programs, including basic output comparison, scripting, source analysis, input and output control, class instrumentation, and using Java reflection. I think we can differentiate these from traditional regression testing as used in industry, in particular because of the necessity of providing detailed feedback on defects to students. We could also address the identification and handling of different types of non-terminating student programs.