Becoming fluent in Java (or any computer language): Overlearning and Active Learning in Introductory Computer Science Learning

NGAI, Grace
CHAN, Stephen C.F.
NG, Vincent T.Y.
Department of Computing
Hong Kong Polytechnic University, Hong Kong

Abstract: It is a well-acknowledged fact that introductory programming courses constitute some of the most difficult courses for instructors in any computer science or engineering program, due to the mix of backgrounds and abilities. In addition, the transition from secondary school to university is not always an easy one for students; changes such as the mode of teaching and style of learning add to students' stress and frustrations. As a result, many of them find introductory programming courses difficult and uninteresting, and some may even drop out of the program as a result. However, as the principles taught in these courses are fundamental to the field, it is important that students are stimulated and challenged, but not intimidated, by these courses.

In this paper, we present our experience with employing the principles of over-learning and active learning in introductory computer science teaching at the university level. Specifically, we discuss our experience with the Introduction to Programming class for Fall 2005-06, in which we introduced new mode on a trial basis. We will present details of how we implemented the course and managed our resources, together with results and feedback that discuss the efficacy of our approach.
Keywords: introductory programming, undergraduate education

Introduction

It is a well-acknowledged fact that introductory technical courses in any computer science or engineering program constitute some of the most difficult courses for instructors. The students often come from a mix of backgrounds and abilities, some of them already quite comfortable and skilled at working with computers while others have barely used a computer for anything other than emailing before. In addition, there is the transition from the very different secondary school teaching style to the university style, which is also not always easy for many students. All these result in feelings of frustration among the students: the weaker students often feel that they are being left behind, while those with more background, feel that they are being held back by their classmates. While feelings of frustration would be problematic for any course, it would be especially serious for an introductory programming course in a computer science department. Since the principles taught in these courses are

fundamental to what the students will be learning over the next several years, it is important that students are stimulated and challenged, not intimidated or bored, by these courses.

In "Seven Good Practices in Undergraduate Education" (Chickering & Gamson 1991), a list of principles which should be observed in any undergraduate class was proposed. The seven principles are to (1) encourage contact between students and faculty, (2) develop reciprocity and cooperation among students, (3) encourage active learning, (4) give prompt feedback, (5) emphasizes time on task, (6) communicate high expectations and (7) respect diverse talents and ways of learning.

While almost all people would agree that these practices are desirable and should be followed, there are also difficulties in implementing them, especially for large technical classes. The sheer number of students makes it difficult for instructors to know each student personally, to begin with. In addition, large numbers of students and tight schedules make it difficult to give prompt feedback, not to mention implementation of active learning and accommodation of different learning styles and abilities. Computer science brings along its own problems, as traditionally, introductory students in CS are not encouraged to work together as the medium makes it far too easy to cheat and plagiarize. The result is that too often, the high expectations are communicated but not picked up on by most of the students.

To address these issues, we propose a change from the traditional style of teaching to a more interactive, workshop-style mode for the introductory computer programming course. This change was implemented in COMP 201, "Principles of Programming", the first and only required programming-specific subject for all incoming first year students at the Department of Computing of the Hong Kong Polytechnic University. We will elaborate on the details of how we implemented these changes, and present results and feedback that demonstrate the efficacy of our approach.

Proposal and Motivations

Traditionally, engineering and computer science courses have taken on a lecture-plus-tutorial format, where course content is delivered in large lectures and then demonstrated to smaller groups in tutorials. Student performance would be assessed using homework assignments, plus one or two midterm examinations during the semester and then a final examination at the end of the semester.

Over the last several years, we had observed that there were consistently cases of students who had successfully passed the introductory programming course, but who would still struggle with writing a program of any size at all. We first tried to address this problem by emphasizing "time on task" and making the course more rigorous and demanding: assignments were added and each assignment was made more difficult. The level of the midterm was also made more difficult. These measures had some effect, but not as much as we had hoped for: we

still saw cases of students passing the course, only to end up in remedial programming classes in a couple years' time with only a cursory knowledge of programming.

Given this, it seemed that a more drastic change was called for. Upon close examination of the course and a general survey of related literature, it seemed that the traditional course structure itself was one of the problems. It has been pointed out (NRC 1996) that the traditional approach, with its large lectures and focus on demonstration rather than participation and investigation, can certainly impart information, but it does tend to turn the learning process into a passive one, where students retain concepts only long enough to get through the course, and then promptly forget about them. Obviously this would be undesirable in any course, but even more so for fundamental ones, which serve as a foundation for higher-level courses. Specifically for us, students who lack familiarity with programming would be expected to have problems with higher-level computer science courses.

In addition, our experience showed that students who did not come in with a prior programming background in secondary school seemed to have a lot of trouble picking up a "feel" for programming. Our hypothesis is that the traditional focus on demonstration over participation does not serve programming classes well. Learning programming, in many ways, is very much like learning a new language; it requires constant use and practice before one becomes "fluent". Traditional tutorial sections do give students a chance to practice what they have learned, but scheduling difficulties often cause substantial time delay between the lecture hour and the assigned tutorial section. This makes the tutorial less effective for the purposes of reinforcing concepts covered in lecture.

As a result of these observations, we set about proposing changes to the course that would address these concerns, as well as better satisfy the Chickering and Gamson principles. The following lists the changes that we instituted and elaborates on the rationale as well as the effect of each change.

Changes in Course Format

The first of our proposals was a change in course format: to move the introductory computer programming course from a lecture-based format to a workshop-based one. Specifically, the following changes would be instituted:

1. The location of the class to move from the lecture hall to a computer laboratory;
2. The traditional weekly lecture plus a separate tutorial section to be combined into a weekly workshop session;
3. Apart from dispensing course content, the workshops would intersperse course content with in-class exercises, which the students would be expected to finish on the spot.

For some of the more motivated students, the physical move of the class location and the additional time were enough to start them experimenting with some of the new concepts covered during lecture. We noticed some of the students cutting-and-pasting the code examples in the lecture notes and trying to compile and run them even without prompting from us; not surprisingly, these students would turn out to be the best-performing students in the class. For the others, the in-class exercises served as a more guided framework that encouraged them to actually type in code and try out the concepts. Some of those exercises were similar to exam questions that ask students conceptual questions on the material which had just been covered (e.g. "If one adds the value of 4 to the character 'A' and saves the result in a character variable, what gets printed out when we do a System.out.println() on that variable?"), while the others are programming exercises which resemble mini-assignments where the students have to type code in from scratch to solve small problems (e.g. "Write a program to ask the user for his weight and height in integers and calculate his Body Mass Index as a floating-point number"). The exercises were supposed to be completed on the spot. Due to limited resources (time and manpower), we could not afford the extra effort of grading these in-class exercises, so a compromise was made: for the written assignments, the correct answers were released to the students once they were done, and as for the programming assignments, the students were encouraged to try their best at solving the problem either alone or through discussion with fellow classmates, but the teaching staff would be on hand to help out if anybody got stuck. Each question carried with it a hypothetical number of "marks", which were made known to the students, who understood these to be the weight that the question *would have* carried if the class exercises had indeed counted for grade. Students were encouraged to grade themselves based on their answers and the correct solution and thus get a sense of how well they were following the content from class.

This change also implemented some of the Chickering and Gamson principles into our class. Some of them were intentional, for example, we had intended to introduce some element of reciprocity and cooperation among our students, and the programming exercises were designed such that they could discuss the solutions and help each other out. Prompt feedback is achieved by releasing the correct solutions immediately to the students, who can use this to assess how well they are doing. Some were unintentional: a side-effect of the programming exercises was that it allowed the teaching staff to have more contact with the students. We had originally assumed that this would be true only for the situations where students needed help with the class exercises, but it turned out that students found it less threatening to ask questions about general course content while their classmates were busy doing their own exercises. Other students, who needed less time to complete the exercises, also took advantage of the increased down time to interact more with us. The result was that we got a better sense of how the class was following the course content.

Another side result of the change in course mode was that it gave the students more leeway to learn in more diverse ways. Students who would have been successful in traditional style classes would still find it easy to learn in this new mode, as we were still presenting concepts in much the same way as we used to do during lecture. However, we noticed that some of the students would often tangent off on their own to further investigate a point that they found puzzling and come back to join the rest of the class when they had satisfied themselves that they had fully understood. The learning-and-reinforcement style of the in-class exercises also softened the entry into university for many of our students, who were used to a similar style from secondary school. (Obviously, the students will have to make the university transition at some point. However, since this is only done for one class in the semester, we feel that it will aid, rather than hamper, their adjustment to university.)

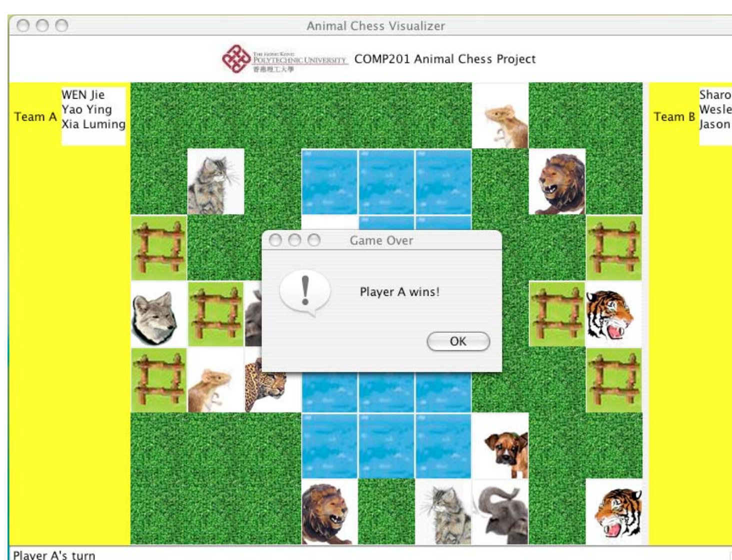

Figure 1: *Animal Chess: The Final Project*

Changes to Assessment Mode

The second of our proposals was a change to the assessment mode. This was in response to another problem that we used to face: the traditional assignments were not effective at assessing students' performance. Over the last several years, we observed that certain students do manage to get past introductory programming courses with only a rudimentary knowledge of computer programming, which would indicate a failure of the evaluation process. Our observations suggested that the assignments were at fault, since it was not conceivable that the students could complete all the assignments independently and yet still not have a grasp of programming. The problem is fairly obvious: there are cases of cheating and plagiarism where the assignments are concerned. Even though there are means of rooting these out, the whole process would take tremendous amount of resources in terms of time and effort, which would be better put to use improving the course and giving struggling students the support that they might need.

To address the problems, we implemented the following changes in the assessment process:

1. Weekly quizzes were introduced in place of the usual midterm;
2. A final project would be added;
3. An optional programming aptitude test was introduced at the end of the semester. Students who chose to take this test would have their term grades determined by their performance on the test, or a weighted combination of their assignments, quizzes and project, whichever was higher.

The format of the old midterm was a written test, essentially an early version of the final exam. We felt that since the final exam would already do a good job of assessing students' conceptual knowledge, we needed a way to assess their programming skills. In addition, we felt that if we wanted students to truly absorb and retain the concepts and skills learned in the classroom, the only way was through overlearning – through constant use and reuse of those same concepts and skills until they became natural. That was the objective of the traditional homework assignments, but the quizzes added another element – since they had to be completed within a given time period, they forced the students to practice writing code on their own until certain things, such as typing out class and method signatures or declaring variables, became almost automatic.

The second item – the final project – was introduced to add an active learning element to the course. Many traditional programming assignments focus on drilling students' knowledge of programming concepts and often illustrate "toy" problems that do not have much to do with the real world, and thus fail to hold students' interest. To address this problem, we introduced a final project that was modeled on the children's game "Animal Chess". We developed the graphical user interface (Figure 1) and the multiplayer capabilities, and the students were asked to design and program the strategy module. The project culminated in a 2-day, head-to-head competition where students plied their programs against each other's to produce a champion. The project related the concepts learned in class to something that the students would be familiar with from their childhood, and by the reactions of the students, it was a success. In the weeks after the project was released, we noticed students bringing chess sets to school and playing against each other in an attempt to work out their strategies; and during the final competition, we observed students frantically modifying their programs on the spot to counter strategies employed by the more successful teams.

The final change – the programming aptitude test – was mainly designed to single out students who were truly good at programming and problem solving, with the side effect of giving the rest of the students a chance to improve their semester grade. The questions on the test were deliberately more open-ended and closer to problem-solving than those in the assignments, and were aimed at singling out students who had been able to internalize their

knowledge of programming and apply it to open-ended, complex problems. As expected, only a few students did well on this exercise, all of them being students whom we had observed to be especially strong programmers during the course of the semester.

Results -- Student Feedback

The new mode of the course was offered on a trial basis during the Fall Semester in 2005-06. The course enrollment was about 120 students. Since the biggest of our computer laboratories had a maximum capacity of 35 students, they were divided into 4 sections. The profiles and backgrounds of the students did not differ significantly from 2004 to 2005.

Table 1 shows the results of the usual semester-end student feedback, comparing the 2005-06 class against the same class from 2004-05. The survey was given as an official (university-level) exercise during the last week of classes and the students were asked to state whether they agreed with the statements. We are aware that this is obviously not the perfect objective evaluation since there are too many variable factors and the question items were not written specifically for the course, but we believe that it does serve as a reliable indicator of the impact of the changes that were instituted.

Table 1: *Student Feedback Survey: Comparison of student feedback for the 2005-06 offering (new mode) of the course against the 2004-05 offering (old mode)*

Question	2005-06	2004-05
Subject matter was difficult	80%	80%
Subject matter was interesting	74%	68%
I was able to understand the subject matter	70%	66%
My interest in the subject has increased	70%	62%
I learned to apply the knowledge to solve real problems	70%	62%

From the figures, it can be seen that about the same percentage of students from both years found the subject matter to be difficult. This is to be expected, as the syllabus was the same and the course material, while not identical, was comparable, and the backgrounds and caliber of the students were about the same. However, there is a marked difference in the way the students responded to the material. Compared with the 2004-05 batch of students, more of them found the material to be interesting (74% vs. 68%), and they were also better able to understand the subject matter (70% vs. 66%). In addition, more of them agreed that the skills they picked up in class could be applied to solve real problems (70% vs. 62%).

On the anecdotal side, we also had some encouraging indicators that the new mode was a success. According to the instructor in charge of “Data Structures and Algorithms”, the second semester follow-up to our course, the 2005 batch of students seemed to have a better grasp of applying practical programming concepts to abstract algorithms and data structures.

Resources

One issue of concern about modifications as drastic as ours is the amount of additional resources required.

Since our mode of instruction combines content delivery with hands-on practice, our classes had to be moved from the lecture hall into the computer laboratory. That created a problem due to the size of the computer laboratories available to us, the largest of which had a capacity of 40. To allow for the possibility of machine failure (experience showed us that at each class, a couple of the computers would invariably have problems of one kind or another), we divided the students into 4 groups of no more than 30 each. This meant that each lecture had to be repeated 4 times, and constituted the biggest demand on resources. However, if we had access to a larger computer laboratory, we could have conceivably fit more students into each section and thus reduced the amount of resources required. In order to provide enough support to all students during the hands-on section, the larger sections would require extra teaching staff to help students who ran into difficulties with the in-class exercises, but this role can be easily filled by teaching assistants or even by senior students recruited as helpers.

The second biggest demand on resources was the addition of the weekly quizzes, which add to the grading load of the teaching staff. This, however, is not as bad as it seems – since the quizzes are designed to be finished within an hour of the students' time, they are also usually quite straightforward to grade and do not substantially increase the amount of resources required.

The rest of the changes do not increase resource demand appreciably. The in-class exercises require time to create, but once the groundwork is done, no extra resources are required. The final project is essentially the same as a regular homework assignment and requires approximately the same amount of effort.

In short, apart from the requirement of providing an environment that would support the in-class, hands-on practice exercises incorporated into the lecture material, the new proposal does not increase resources appreciably.

Conclusions

This paper describes and motivates a set of proposed changes to an introductory programming course, which was implemented on a trial basis to a class of 120 students in the 2005-06 academic year. The changes were proposed to address some shortcomings which had become evident in previous years and were aimed to better enable students to pick up and retain fundamental concepts. Overlearning was employed to constantly reinforce programming concepts in the minds of the students. To better relate these fundamental

concepts to real problems, active learning was employed at an early stage in the form of a project. In addition, the changes also had the effect of better accommodating different learning styles and encouraged students to be more proactive in their learning. The usual end-of-semester feedback showed that the changes were well received by the students and anecdotal evidence shows that the changes were successful in helping the students to better understand and retain the concepts taught in class. An analysis of the resources required shows that once the problem of a suitable environment that can support the new mode of interspersing content delivery with hands-on exercises is addressed, the increase in required resources is not appreciable.

References

Chickering, A.W., and Gamson, Z.F. (1991). Applying the Seven Principles for Good Practice in Undergraduate Education. *New Directions for Teaching and Learning*, 47, 1991. San Francisco: Jossey-Bass Inc.

NRC, From Analysis to Action: Undergraduate Education in Science, Mathematics, Engineering, and Technology, National Academy Press, Washington, DC, 1996.