Programming Studio: A Course for Improving Programming Skills in Undergraduates

Michael Woodley
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
(217) 244-1971
mwoodley@cs.uiuc.edu

Samuel N. Kamin
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
(217) 333-7505
kamin@uiuc.edu

ABSTRACT
Even after taking numerous programming courses, many students have poor programming skills. This is a problem not only in their post-graduation employment, but even in the higher-level Computer Science courses, where large programs are routinely assigned. Yet, teaching programming skills is expensive; like teaching writing, it can only be accomplished by a repeated cycle of writing, getting informed feedback, and rewriting. In this paper, we describe a computer science course designed around the concept of a studio course like those used in art and architecture. Its key elements are practice, public presentation, and review by peers in a small group. We discuss our experience in teaching the course for two years. We believe this course can be replicated and taught, at reasonable cost, even in large CS departments.

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

General Terms
Measurement, Design, Experimentation.

Keywords
Programming, practice, undergraduate education.

1. INTRODUCTION
Computer Science departments strive to teach their students both the *fundamentals* of computers–computer architecture, algorithmic analysis, etc.–and the *skill*–some would say art–of programming computers. In many cases, the skills are learned only as a side effect of learning fundamentals–by doing homework assignments that require programming in classes that are primarily teaching fundamentals. Normally, only introductory courses are actually *about* programming. In modern Computer Science departments, skills teaching is denigrated, the attitude being that skills can be learned on the job.

This approach has several unfortunate consequences:

- *It is uncertain*. Since most skills acquisition occurs in courses in which "skills acquisition" appears nowhere in the syllabus, it is entirely up to the instructor to decide how much emphasis is placed on it. Students may graduate without ever having taken a course in which the instructor took this part of the course seriously.
- *It is poorly done*. For the same reasons as just cited, instructors rarely put much emphasis on teaching skills. An instructor in a high-level CS class is unlikely to put a lot of lecture time into discussing how the programming homework should be done, but instead will spend nearly all the lecture time explaining the algorithms. More importantly, the *grading* of programming assignments often consists of little more than testing the program for correct functional behavior, providing little feedback on the program structure or the student's programming style.
- *It is late*. At best, the student will *graduate* with decent skills obtained by writing many large programs over several years. But in the meantime, the students will have struggled with those programs, distracting him from the very fundamentals to which the courses are devoted.

In our department, we decided several years ago that it would be beneficial for our students to *learn to program* in a course dedicated to that goal. The course would come after the students had learned the basic concepts of programming but before the more advanced courses.

This decision left us to confront the fundamental problem in teaching students how to program: how to provide individualized instruction in a department with about 800 undergraduate CS majors. It is our belief that learning to program well is much like learning to write well: the student needs to receive detailed feedback, rewrite, and receive more feedback. Yet meeting with students individually would entail an impossible time commitment, and providing carefully written comments on students' programs is not only time-consuming but exhausting. With sufficient resources, these methods would be feasible – after all, this is how writing is taught, even on a large campus like ours. However, in CS, as noted earlier, the *craft* of programming is not given its full due, and these resources are not available.

Our solution was to create a course that borrows certain elements from studio courses in art and architecture, above all, the aspects of *practice* combined with *public performance*. In those courses, students all work in a common area, and since their work is primarily visual, it is under constant scrutiny. Periodically, there are reviews (or "critiques," or "pin-ups"), in which the students present their work to a panel of professors (with other

students often in the audience as well). In our course, we have tried to reproduce the spirit of those studios.

In this paper, we describe our experience with this course. Section 2 explains the structure of the course in detail, and section 3 says why we believe it is succeeding. Although the basic structure has not changed since we introduced the course, we have made numerous adjustments to make the class more productive; to give readers the benefit of our trials and errors, we discuss some of these adjustments in Section 4. Section 5 discusses some ongoing challenges and future plans.

1.1 Related Work

Studio-style courses are commonplace in departments of fine arts, creative writing, and architecture. Their essential feature is the intensive, on-going discussion of one's work, both with the instructor and with classmates. The idea of using a studio approach to teaching programming has been championed by Richard Gabriel [Gab].

Perhaps due to the instructional manpower requirements, such courses have not became popular in Computer Science programs. There are some exceptions. Carbone and Sheard [CS02] offer an introductory programming course at Monash University in Australia that is explicitly based on architectural studio courses (the "Bauhaus" model). Clancy et al. [CTRSL03] investigate a very different model of cooperative Computer Science education at UC Berkeley, in which group studio projects largely replace traditional lectures, and the instructor's role becomes that of a consultant. Gonsalvez and Atchison [GA00] report on their use of the studio method in a set of IT design courses at Monash University, taking note of the heavy resource requirements of such courses. A more ambitious effort is reported by Docherty et al. [DSBK01], who have structured an entire CS curriculum at the University of Queensland, Australia, around studio-style courses (again with explicit acknowledgement of the fine arts inspiration).

Our studio course differs from these courses in several ways: It is aimed at intermediate, rather than beginning, students. Also, it is a single course, not a curriculum, making it more easily reproducible in departments with standard curricula. There are several courses being offered by Computer Science departments that are closer to ours. Tomayko [Tom87] reports on a studio course he designed for the Software Engineering Institute at Carnegie-Mellon. The Programming Practicum course at Harvey Mudd college is structured around the annual ACM Programming Contests; this is a great motivator, and the college has indeed done extremely well in that competition. The Software Practicum at Georgia Tech (CS 2335) is like our course, frankly devoted to teaching programming skills, although the topics tend toward "programming-in-the-large" topics, such as UML-based program design.

1.2 Contributions of This Work

The contribution of this paper is the description of a course that we have taught for the past two years at the University of Illinois, which provides a high level of individualized instruction in programming to a large number of students at a reasonable cost. We discuss the lessons we have learned in teaching the course and the adjustments we have made, as well as areas of continuing concern and plans for the future. We believe the type of course we have introduced can be duplicated anywhere with a modest commitment of resources.

2. COURSE STRUCTURE

The Studio course has no fixed syllabus. It meets in a one-hour lecture once a week and each students attends a two-hour discussion section each week. Programming assignments are usually specified less precisely than in ordinary low-level CS classes, and usually last two weeks; we have recently added a four-week final project devised by each student (with the instructor's approval). We discuss these three components–lectures, discussions sections, assignments–in the following, and also discuss the cost of the course.

First, we need to say something about where this course falls within our curriculum. The question of where the course can be most profitably placed is still being debated, but we have chosen to place it just between the initial set of programming courses and the upper-level courses. Specifically, students in the Studio will have taken CS1 (using Java), CS2 (using C++), a course in a computer architecture, and a course in systems programming, all of which are programming courses.

We have considered moving the studio earlier–even to just after CS1–but have not actually experimented with any other placement. With the current position, students come to the course having learned the main techniques of programming (including concurrency), at least superficially. Thus, in the Studio, they can practice all these techniques. The Studio helps students integrate lessons from all their previous classes.

2.1 Lectures

The instructor lectures once a week, either on a topic of general programming interest or on the next assignment. Thus, lectures might present material on:

- Proper programming style – the use of comments, code formatting, choice of variable names, etc. Good style would have been taught in earlier classes, but probably was not consistently enforced.
- Programming tools – By "tools" we mean debuggers, version control systems, "makefiles", and such. Again, students may have been introduced to these tools in earlier classes, but at a time when they were heavily occupied with trying to understand the other, more conceptual, course material.
- Algorithms and technologies needed in programming assignments.

Almost every assignment introduces some technology – an API or programming language – with which the students are not expected to be familiar. The lecture gives them a start at learning the technology, although they will still need to spend time on their own to learn it well enough to do the assignment. With new assignments being given every two weeks, topics like these account for about half the lectures. We will sometimes have guest lecturers, possibly members of our own faculty.

2.2 Discussion Sections

The weekly two-hour discussion sections are the focus of the course, its major innovative feature, and the aspect of the course on which we have expended the most effort. Sections consist of five students and an instructor who acts as moderator. Each

student is given 20-25 minutes to present their week's work. The students present and explain their programs. The instructor and other students ask questions and critique the student's work.

The public presentations have several effects:

- Students will work hard during the week to avoid looking foolish in their presentation.
- Students will naturally drift toward writing code that is easy to explain – which is a focus that nearly always produces better code.
- Students cannot cheat. If a student gets code from someone else, this fact will be exposed as soon as they start to explain it.

The last of these effects has an important corollary. In other courses, instructors are sometimes reluctant to have students work in groups because they want to be sure every student learns to program individually. In the Studio, we are sure that each student is learning; this relieves the pressure on the other courses and permits them to use modalities – team projects, pair programming, guided lab work – that are deemed preferable on most aspects *except* the guarantee of individual work.

2.3 Assignments

The assignments are intended to provide the students with ample opportunity for creativity. They are not extremely difficult, but generally require a considerable amount of time and coding. This is in keeping with the philosophy of the course which is to provide an environment for practice. Here are some example assignments:

1. **Raster Analyzer**. We begin each semester with a simple program to review basic programming style, such as this one:

   > Read in grid data from a file, compare every element in the grid to its eight neighbors and generate a byte for each member with a bit set for each neighbor that is identical. Write your resulting grid data to a file. Create your own input data in whatever form you find convenient. Write in C++.

   In the second week, we have students rewrite this program, applying *low-level* optimizations; students have taken courses that should make this exercise straightforward, but they have a surprisingly weak concept of precisely what computation is produced by a given program, so we find this exercise extremely useful.

2. **Machine Portfolio Generator**. Students build the infrastructure for an online portfolio of their work (which we have sometimes used as the presentation format in discussion sections as well):

   > Read a text file that describes the format of a web-based portfolio, including names of source files and mark-up information. The resulting web pages contain descriptions of projects, with source code, documentation, etc. Use any language you like.

   The portfolio assignment is a multi-week assignment which allows for a considerable amount of creativity and can be built upon in numerous ways. We have used it to introduce XML (for the layout specification), PHP, and MySql – the kind of practical topics that are often omitted from the more conceptual courses.

3. **Final Project**. Students propose a project for the remaining 4 weeks of the semester. Their proposals must indicate what they plan to learn, and give a plan of work specific enough so that we can determine their expected workload.

2.4 Cost

Our course has one instructor, a number of teaching assistants (graduate students) and studio aides (undergraduates). The mix depends upon the availability of TAs. We have found that studio aides drawn from the best students in previous semesters are usually excellent. The undergraduates and TAs need to be selected based on their programming skills, their ability to constructively offer criticism, and their ability to foster a constructive atmosphere among the students in the section.

In terms of manpower, we have taken the view that no one can be asked to moderate studio sections for more than 10 hours (5 sections) per week. Thus, each graduate teaching assistant can teach 25 students. This means that, on a per-student basis, the studio employs about twice as many TAs as an ordinary course. On the other hand, the course requires little work of the TAs outside the discussion sections, as there are no exams and no additional homework to be graded. (The TAs need to be available to students who need more help than is provided in discussions, which will add several hours per week to the workload.) Thus, it is not unreasonable to find part-time work for these TAs in other courses, as a way to lower the overall cost to the department. Employing undergraduates also lowers the cost. Undergraduates are paid only for the time they spend in the discussion section and they do not offer office hours or additional instruction.

Since the class is now required, it will serve 100 students per semester, requiring four TA's. This is a large number for a course of this size, but our department currently employs upward of 80 TA's, so this is a financial burden we can bear.

3. EVIDENCE OF SUCCESS

In addition to testimonials from students indicating that they found the course helpful, last semester we asked students to write their first assignment again at the end of the semester. This rewrite of the first assignment was required but not graded. We were curious to see if there was actual improvement in coding. To some extent, improvement is a subjective thing. However, when looking for specific habits like the use of constant definitions instead of "magic numbers", good variable naming, use of functions and prototyping, commenting and readability, we can actually see improvement. As one might expect, students with more prior experience show less improvement. Experienced students stick with the habits to which they have become accustomed whether those are bad or good habits.

For example, in one inexperienced student's first version of the assignment (raster analyzer), the entire program was in main(), "magic numbers" were used, boundary conditions were checked at every access, output and processing were mixed, and there were few comments. In the second version, he had moved code into functions which were commented; magic numbers were gone, variables and functions were named well, and performance improvements discussed in lecture had been used.

4. LESSONS LEARNED

The weekly discussion sections are the key educational component of the Studio course. As described earlier, in these, every student – five in each section, meeting with an instructor – gives a 20-25 minute presentation about his or her program. Students critique each other's work, with the instructor acting as moderator.

Before we began giving the course, we were afraid that students might be overly harsh in their critiques. This has not proven to be the case. Rather, it is often the instructor who offers the most cogent criticism, even in cases where the flaw in the presenter's code is obvious. Undoubtedly, one reason for this is that students simply don't recognize the flaws. However, we believe the main reasons are that (a) students are not confident enough in their judgments to make criticisms that may turn out to be unwarranted, and (b) students do not want to appear to be mean to other students. Students are discouraged from making trivial criticisms. More experienced students are encouraged to offer suggestions of better approaches to avoid harsher forms of criticism.

Rather than give a complete history of discussion section structures we have used, we divide our efforts into several categories and discuss the range of differing methods for each. We then describe the current structure, which seems to work well.

Presentation. We feel it is important that student *prepare* their presentations carefully and not simply bring their latest code and discuss it extemporaneously. At the same time, we want the presentation to focus on the *code itself*, not on functional behavior or visual appearance or idealized pseudo-code. In attempting to forge the best compromise, we have tried several approaches:

- *PowerPoint presentation*. The problems with this approach are that only a small portion of code is available for examination and students spend too much time on preparing the presentation.
- *Programming portfolio*. The portfolio is web-based and contains a page for each project. On each page is a description and a link to the source code files written for the project.
- *General code review*. Students use code editors of their choosing. One advantage is that many editors have syntax highlighting which makes the code easier to read.

Discussion. Initially, everyone in the discussion section was expected to see the code and make an examination of the code "cold". That is, they see the code for the first time during the discussion section. This puts everyone on the same footing in regards to familiarity with the code; however, it is difficult for students to understand the code well enough to ask meaningful questions in the time allotted. We have tried several alternatives:

- *All participate*. The first approach was that everyone was expected to ask questions of each of the other presenters. The questions asked tended to be easy. This is due both to students wanting to be easy on each other in hopes that they would receive easy questions and to students giving a cold read of the code. Some students sat quietly.
- *Designated questioner*. We changed the expectation to be that one student would be the designated examiner for another student, so that by the end of each session each student had presented and been the primary investigator for another's code.
- *Code available beforehand*. We now expect students to hand their code in the day before discussion section so that the other students can examine the code. Students did not put in the time reading all of the other programs.
- *Code available beforehand with one specific investigator*. We have put the best methods together so that students are now only expected to be familiar with the code of one other presenter on the day of discussion. All students are still expected to participate but one is given a lead role in examining the code of another.

Student assessment. The TA keeps track of the participation of each student using a questionnaire that is specific to each assignment. Originally, grading was a count of check marks on that questionnaire. We have refined that so that each question is weighted. The expectations in the questionnaire are provided to the students in advance so they know what they will be judged on in class.

At present, we are employing a discussion structure based on the roles outlined for code walkthroughs in the book Code Complete [MCC01]. There are 4 roles in this scheme: author, moderator, scribe and reviewer. The TA is the moderator and the students take turns at the other roles.

Students now hand in their code the day before discussion sections start for the week. All sections, regardless of when they meet turn in their code at the same time. The TA designates the primary reviewer and emails a copy of the author's code to the designated reviewer. The primary reviewer familiarizes himself with the author's code. In discussion section, the author gives a presentation of his code and focuses on sections that presented problems, were tricky or that he finds particularly interesting. After his short presentation, the remaining students in the discussion section ask questions about the code.

The designated reviewer is expected to direct attention to code that is particularly interesting, difficult or that needs improvement. The scribe records improvements that are indicated during the discussion.

The TA acts as moderator by keeping things on track, asking questions that he believes are pertinent and insures the list of obligations is fair. The next week the TA uses the list of obligations as a checklist to see if the author made the changes indicated.

5. CHALLENGES AND FUTURE WORK

We have not yet given the Studio course to the full complement of 100 students. Though we do not have any concerns about having adequate manpower, we are worried about quality control. The course is a lot of work for the students, and the discussions can be stressful. In this environment, incompetent or insensitive instructors, and personality conflicts among students, can have an exaggerated and highly detrimental effect on some students. We anticipate instituting a formalized system of TA training, review, and oversight, possibly including taping discussions sections to critique the instructors.

We need to continually work on maintaining the quality of the discussion in the discussion sections. We have several ideas for refining the structure: requiring students to make the next week's improvements to someone else's code; using software to enhance the interaction of students; and allowing students to work in teams on a project.

We are currently working, under a grant from Microsoft, to introduce new technology into the studio discussions. We have developed a system for doing code reviews on networked Tablet PC's, which supports the "roles" mentioned in section 4. We hope this system will improve both the quality and recall of discussions, as well as helping with grading (which has an indirect effect on discussion quality, as students are graded on class participation).

Some other ideas for the Studio that have not yet tried are:

- Invite faculty to design programming assignments around their own research – possibly incorporating software they have written – and give the lecture on that assignment.
- Draw assignments from Source Forge or some other open source repository.
- Read programs. We would like to find a set of exemplary programs to review in lecture, as well as on the first day of discussion.
- Assign different criteria for some assignments, e.g. conciseness, generality, speed, etc. Implicitly, our basic criteria of program quality are clarity and simplicity, and we feel comfortable with this. But having students write to other criteria – even when those do not normally represent desirable characteristics in the real world – can enlighten by highlighting the boundaries of programming practice.

6. REFERENCES

[CS02] A. Carbone, J. Sheard. A studio-based teaching and learning model in IT: what do first year students think? *ACM SIGCSE Bulletin, Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*. 34, 3, June 2002.

[CTRSL03] M. Clancy, N. Titterton, C. Ryan, J. Slotta, M. Linn. New Roles for Students, Instructors, and Computers in a Lab-based Introductory Programming Course. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education.* Reno, Nevada. 2003. 132 – 136.

[DSBK01] M. Docherty, P. Sutton, M. Brereton, S. Kaplan. An Innovative Design and Studio-based CS Degree. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*. Charlotte, North Carolina. 2001. 233 – 237.

[Gab] R. P. Gabriel, Master of Fine Arts in Software. Published online at http://www.dreamsongs.com/MFASoftware.html.

[GA00] C. Gonsalvez, M. Atchison. Implementing studios for experiential learning. In *Proceedings of the Australasian Conference on Computing Education.* Melbourne, Australia. December 2000. 116-123.

[MCC01] McConnell, Steve. *Code Complete*, pages486-487. Microsoft Press, 2nd Edition, 2004

[Tom91] J. E. Tomayko. Teaching Software Development in a Studio Environment. In *Proceedings of the Twenty-second SIGCSE Technical Symposium on Computer Science Education*. San Antonio, Texas. 1991. 300 – 303.