A Course on Programming and Problem Solving

Swapneel Sheth
Christian Murphy
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
swapneel@cis.upenn.edu
cdmurphy@cis.upenn.edu

Kenneth A. Ross
Department of Computer Science
Columbia University
New York, NY, USA
kar@cs.columbia.edu

Dennis Shasha
Department of Computer Science
New York University
New York, NY, USA
shasha@cs.nyu.edu

ABSTRACT

At its core, Computer Science is the study of algorithmic problem solving. Although it is necessary to teach programming, data structures, computer organization, etc., students should ultimately learn to use these things to solve problems, understand what is good and bad about their solutions, and share their solutions with others.

This paper describes a course that focuses on the four steps of the problem solving process: algorithmic thinking, implementation, analysis, and communication. This course, based on Knuth's popular seminar at Stanford, has been extremely successful at the authors' three institutions. In addition to discussing the course's objectives and methodology, we present sample problems, summarize the outcomes and feedback from students, and give advice to other educators looking to create a similar course.

Keywords

problem-based learning; open-ended problems; soft skills

1. INTRODUCTION

Although modern Computer Science curricula focus on programming, data structures, computer organization, software engineering, etc., we must not lose sight of the fact that the goal of CS is to use algorithms to solve problems, and that problem solving in CS is a collaborative activity that involves analyzing and communicating solutions, not just implementing them.

This paper introduces a unique course that focuses on problem solving in CS. This course, based on Knuth's popular seminar in the 1970's and 80's, has been taught at the authors' three institutions for over 15 years, and develops students' problem solving skills using techniques that they have learned during their CS training. In the course, students work together to use programming techniques to solve open-ended problems from the domains of optimization, simulation, etc. There are no "correct" answers to these problems; rather, the focus is on the four steps of the problem solving process: algorithmic thinking, implementation, analysis, and communication.

This paper describes the course's objectives and methodology, presents a sample problem, summarizes the outcomes and feedback from students, and serves as a roadmap to other educators looking to create a similar course.

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA

DOI: http://dx.doi.org/10.1145/2839509.2844594

2. HISTORY

Donald Knuth started teaching CS204, "A Programming and Problem Solving Seminar," at Stanford in 1976. This class was designed to be taken by new CS PhD students during their first quarter. He taught the class roughly every two years, and transcripts of those classes can be found as Stanford Technical Reports (e.g., [4, 21]). Video of the 1985 class can be found at http://scpd.stanford.edu/free-stuff/engineering-archives/donald-e-knuth-lectures under the heading "Aha Sessions."

In the preface for the 1976 class report [4], Knuth states,

> The purpose of CS 204 is to teach skills needed for research in computer science, as well as programming skills. I assigned five programming problems drawn from different areas of computer science; none of these problems was easy, but they were all intended to be sufficiently enjoyable that the students would not mind working overtime.

The main features of the class were to:

- Work on open-ended problems.
- Work on several different types of problems.
- Use programming as a way to gain knowledge about a problem (and potentially solve it).
- Have fun working on challenging problems.
- Work in small groups.
- Openly discuss approaches with students doing most of the talking; the instructor serves primarily as moderator.
- Collaborate (building on one another's ideas as the class progresses through a problem) rather than compete (keeping one's ideas private until the end).

In 1999, the third author, who had the privilege of being Knuth's teaching assistant at Stanford, created a new course at Columbia University based on the Stanford model. The

course was aimed at both undergraduate and graduate students and was called "Programming and Problem Solving." A transcript of the first instance of the class is available [17]. As of summer 2015, the class has been offered fifteen times at Columbia University. The first two authors took this course as graduate students at Columbia University in Fall 2006 and Fall 2004, respectively. After joining the faculty at the University of Pennsylvania they introduced this class there in 2013 and as of this writing it has been offered five times.

A similar class, "Heuristic Problem Solving," has been taught at New York University since 2001. While writing a book on great computer scientists [18], the fourth author revisited some of the problems Knuth had used at Stanford. Recognizing that students learn best when they do rather than when they sit and allegedly listen, the Heuristic Problem Solving course (taught for the 16th time in Fall 2015) is structured around simple-to-state but hard-to-solve puzzles, many drawn from the fourth author's columns in Scientific American and Dr. Dobb's Journal.

3. COURSE DETAILS

In this section, we describe in detail the course's goals and educational objectives and the overall course structure.

3.1 Educational Objectives

Computer Science can be thought of as an algorithmic problem-solving activity in which we create, implement, analyze, and communicate solutions. The main objective of this course is to strengthen students' abilities in these four areas, specifically by having them do the following:

- Identify appropriate algorithms, data structures, etc. to use in order to solve a problem.
- Apply programming skills as a means of implementing an algorithmic solution.
- Develop intuition to enable students to come up with creative approaches to problems.
- Reuse and modify code.
- Collaborate with other students on programming projects.
- Develop skills for analyzing solutions.
- Incorporate feedback from others in order to enhance the quality of the solution.
- Improve communication skills (group discussion, written, presentation, teamwork).

More formally, using Bloom's taxonomy of educational objectives [1], most CS courses focus on the lower two levels, i.e., "knowledge" and "comprehension," with partial coverage of the third level, "application." This course, on the other hand, assumes that knowledge and comprehension have already been gained in a variety of CS classes. It thus focuses primarily on the upper levels of the taxonomy, namely "analysis," "synthesis," and "evaluation." Although capstone and project courses often touch on these particular aspects, there are few, if any, classes with these specific objectives in a traditional CS curriculum.

3.2 Intended Audience

This course is appropriate for upper-level undergraduates and graduate students, and serves as a capstone experience for them in that it builds on what they have learned in terms of algorithms, data structures, and programming and requires them to apply those as "tools" in solving problems.

In terms of prerequisites, data structures, algorithms, and at least two years of programming experience are required. Upper-level courses like Artificial Intelligence and Machine Learning are recommended.

3.3 Course Structure

There are no lectures in the course; rather, the course uses the Problem-Based Learning [20] methodology throughout the semester. The course consists of four cycles, each lasting approximately three weeks. In each cycle, the instructor first presents the problem to be solved. All the problems are open-ended and there are no "correct" solutions. We describe an example problem called "Organisms" in Section 4.1.

For each problem, the instructor provides a simulator that implements the rules and logic and that allows students to evaluate their solutions and/or compete against other teams' implementations in a variety of configurations. Students program to a given interface as part of their solution. The students can thus focus on the implementation of their approach/algorithm and not worry about implementing the details related to simulating the problem itself.

After the initial discussion of the problem, teams are formed (details below) and students work with their groups outside of class time. Subsequent class meetings consist of discussions of the students' insights into the problem and the work they have done in their group. There are a series of team "deliverables" due over the course of the problem cycle, and each group demonstrates their progress to the rest of the class in each class meeting.

At the end of the problem cycle, a tournament is held to determine the "best" solution, by an agreed-upon definition of "best," for a variety of configurations according to the problem. Although tournament results are not necessarily related to the students' grade, they are indicative of the quality of the solution and serve as a motivating factor.

3.4 Class Size and Teams

Due to the nature of the course, class size is typically limited to 25–30 students. Teams usually consist of three or four students, thus resulting in six to eight teams overall. For each problem cycle, new teams are created and group membership necessarily changes. The constraint that we add is that two students cannot work together more than once, as long as feasible. This gives everyone an opportunity to work with different people, leading to a greater diversity of ideas and approaches.

3.5 Open Source Nature

What perhaps makes this class even more unique compared to others typically offered in most CS departments is the open source nature of the class. Teams are allowed (and encouraged) to take ideas and even code from other teams, as long as they are appropriately attributed and cited. This allows for a freer exchange of ideas in the class discussions and it is very common to have a team say: "We really liked what team X proposed and since they were pursing another approach, we decided to implement their idea this weekend."

3.6 Communication and Presentation

An important pedagogical goal of this course is to improve students' communication and presentation skills in technical

subjects. Towards this end, at the end of each problem cycle, every team submits a final report, which includes details of the team's solution, critical analysis of their approach, an overview of the implementation, and analysis of the tournament results. Every team also does a short presentation in the last class for each cycle.

3.7 Evaluation and Grades

For each problem, the grade is based on the students' implementation and deliverables, including:

- the novelty of the approach
- the thoroughness of the report
- the clarity of the report presentation
- the quality of the code
- the correctness and generality of the proposed solution
- the efficiency of the proposed solution

A large part of the course grade is also based on class and group participation. As such, class attendance is mandatory and missing several classes will result in a significant drop in the grade. Likewise, (for some of us), students evaluate each other at the end of each cycle, and these evaluations are used in calculating the final course grade.

4. SAMPLE PROBLEMS

One of the main contributions of this paper is to provide a large repository of existing problems that can be used by instructors at different institutions. There are three different sets of problems maintained by authors at the three institutions. There are over 85 problems available in total; for all the problems, software support (code, simulators, and GUI) is available either on request or directly through our website at http://programming-and-problem-solving.github.io/.

4.1 Organisms

We now describe an example problem called "Organisms." This problem has been used several times and is one of the most popular as voted by the students. The description below is the handout given to the class at the start of the problem cycle.

Consider an electronic world consisting of an m by n grid. Virtual "organisms" can exist on this grid, with an organism able to occupy a cell on the grid. Organisms have energy that can be gained or lost in a variety of ways. When an organism runs out of energy it dies, and vacates the cell it formerly occupied. An organism can have at most M units of energy. An organism may do one of several things during a virtual time cycle:

- Move one cell horizontally or vertically in any direction. The world wraps, so that an organism traveling off the right edge of the grid appears on the left edge, and similarly for the top and bottom edges. A move uses some energy.
- Stay put and do nothing. This move uses a small amount of energy.
- Reproduce. An organism can split in two, placing a replica of itself on an adjacent square. Each resultant organism has slightly less than half of the initial energy of the original organism since reproduction costs some energy.

An illegal move (such as trying to move or reproduce onto an occupied square) results in a "stay put" outcome.

There will be food scattered over the grid. One unit of food corresponds to u units of energy. Food reproduces according to the following rules:

- An empty square has a small probability p of having a single unit of food "blow in."
- For cells already containing food, but no organisms, every food unit on a nonempty square has a small probability q of doubling. This doubling is independent of other food units on the cell. So, a cell with three units of food may have anywhere between three and six units of food on the next cycle. Nevertheless, no cell may have more than K units of food at any one time, due to space constraints.
- Cells with organisms never obtain additional food. (The organisms block the light.) In fact, an organism that is on a cell with food, and which has energy no larger than $M - u$, will consume a unit of food and add u units of energy to its store. If an organism is hungry, it can eat one unit of food per cycle until either the food runs out, or it achieves energy greater than $M - u$.

An organism has an *external* state that is an integer between 0 and 255. This state may be changed by the organism during the course of the simulation. The external state is visible to other organisms, as described below.

An organism can "see" in the four orthogonal directions. An organism gets information about:

- Whether there is food or not on a neighboring square (but not how much food).
- Whether there is another organism on a neighboring square. If there is, then the external state of the neighboring organism is also available.
- The amount of remaining food on the organism's current cell.
- The amount of energy currently possessed by the organism.
- Values of the simulator parameters s (energy consumed in staying put), v (energy consumed in moving or reproducing), u, M, and K, but not p, q, m, or n.

An organism's "brain" is a program that you will write. Since there will be many organisms on the grid simultaneously, each will run a separate instance of the brain code. Each instance will have access only to the local environment of the organism. Organisms are placed randomly on the grid, and don't know their coordinates. The brain can keep a history of local events for the organism if it's useful.

Organisms cannot identify their neighbors. Neighbors may be of the same species (i.e., have the same programmed brain), or of a different species (i.e., have a different brain). This will be important for simulations in which multiple organisms from different groups are placed on the same grid. (How might you use the external state to help in identification? What about impostors?)

Organisms act one-at-a-time from top-left to bottom-right, row by row. We number the top-left cell as (1,1) and the bottom-right cell as (m, n). That means that the state of the virtual world seen by an organism at (x, y) reflects the situation in which all organisms in positions (x', y') lexicographically less than (x, y) have already made their moves,

while organisms in positions lexicographically after (x, y) have not yet moved. This convention allows all operations to happen without any need for resolving conflicts between organisms (for example trying to move to the same cell). However, it leads to some slightly unintuitive effects:

- An organism that is sensed to the west is usually in its final position for the cycle (can you think of an exception?), while an organism sensed to the east may or may not be in its final position.
- An organism moving east can be sensed from the west, while an organism moving west cannot be sensed from the east. (There's an exception to this observation: what is it?)

We'll provide an organism "simulator" that reads in one or more organism brains, places one organism for each such brain randomly on the grid, and lets the organisms behave according to their brains' instructions.

There are several goals for this problem:

1. Your organism should be able to survive and replicate in an environment where it is the only kind of organism present. This may not be as easy as it sounds. Overpopulation may lead to consumption of all the food. If p is sufficiently small, extinction may ensue. The goal is to achieve the highest long-term stable population.
2. Your organism will be tested in environments containing other organisms. The goal here is primarily to survive, and secondarily to survive in higher numbers than competing organisms. Can your organisms populate the grid faster than their competitors? (Is that even the right strategy given the possibility of everybody going extinct?) How might you program your organisms to "recognize" different organisms based on their state and/or behavior? If you can distinguish members of your species, how might you behave differently to members of another species? Your goal here is not necessarily to obliterate other species, but to maximize your fraction of the population.
3. In the discussion so far, we haven't limited the size of organisms' brains. How might your programs change if (to simulate an organism's limited brain size) we limited the complexity of the brain code? For example, what if we eliminated all looping constructs (while, for, etc.) from brains?

5. OUTCOMES

This course has been taught several times at the three institutions: the first two authors have taught this course five times over three years at University of Pennsylvania; the third author has taught this course almost every year since 1999 at Columbia University; the fourth author has taught this course thirteen times at New York University.

The student experience at all three institutions has been overwhelmingly positive. Overall course quality scores range between "Very Good" and "Excellent." We now describe the qualitative feedback from the various offerings of this course. The feedback comes from a variety of sources including end of semester wrap-up sessions, official university evaluations, and email sent to the authors. The feedback is grouped into the following categories:

Overall Course: This course has consistently been very highly rated and has been very popular with students. Selected comments include: "One of the best courses I've taken."; "I would not only recommend this to any CS student, but would demand that they take it, even make it a core course of the program."; "It is the most unique course in the department."; "This is the best thing that happened to me here. The course structure is great, the problems were well chosen."

Course Structure: The open-ended nature of the problems was very appealing to students. Selected comments include: "These open ended problems excited an interest and an enthusiasm that I've never seen in almost any other class."; "The open-ended nature of the problems really helped to develop my skills of independent problem solving."; "This also changed my idea of how a course SHOULD be taught—namely using competitions and open-ended projects as motivating devices."

Course Workload: Students felt that the course entailed a lot of work but that it was worth the effort. Selected comments include: "This is the hardest, best class I have ever taken. It made me weep it was so hard, but I never have learned more or had so much fun as in this class."; "The work was VERY heavy, but was reasonable considering the nature of the class."

Teamwork, Discussion, and Communication: Students mentioned that a strong positive aspect of the course was the emphasis on teamwork and presentation and communication skills. Selected comments include: "The class discussions really helped open me to new perspectives on the problems, and the regular presentations helped develop my skill in communicating technical matters."; "The open discussions in the bi-weekly classes definitely facilitates the evolution of new ideas as students need not worry about their own plans being copied by other teams – in fact one wishes that his or her ideas are borrowed by the participants as credit will be given to them."; "It's also a class where teamwork is key and good communication is vital."

Class Size: As class sizes are growing at most universities, students found the small class size in this course to be a very unique and beneficial aspect. Selected comments include: "Having a small class is a wonderful thing."; "The small class size and that too of hand-picked students really made the discussions to be of very high quality."; "Having fewer people in the class makes me more comfortable and gives me the chance to speak my mind. Something I have not had in ANY class up to this point."

While there has been occasional negative feedback from students, it almost always concerns being in a group with a team member who was not willing to pull their weight. Because groups switch after every problem cycle, nobody is paired with the same partner more than once.

6. RELATED WORK

This course builds on and complements existing work on flipped classrooms and active learning, project-based courses, simulation-oriented games, and integrating soft skills in Com-

puter Science classes. We summarize the important related work in this section.

Problem-Based Learning (PBL) and the inverted/flipped classroom [14] have become popular pedagogical tools. In PBL, appropriate problems are used to increase knowledge and mastery of the material. Group learning is a key part of PBL and the goal is not "only the acquisition of knowledge but also several other desirable attributes, such as communication skills, teamwork, problem solving, independent responsibility for learning, sharing information, and respect for others" [20]. Using aspects of PBL, in the inverted/flipped classroom, activities that traditionally happen in the classroom (such as lectures) happen outside the classroom and vice-versa. Students review slides before coming to class and class time is used to review the material and practice problems. This approach has been been used successfully in a variety of CS classes. Lockwood and Essenstein [15] describe using an inverted classroom to teach introductory programming. Gehringer and Peddycord [7] use an inverted classroom to teach computer architecture. Kurtz et al. [13] describe using tablets for active learning in lectures. Our course leverages and complements all of this prior work in that learning occurs outside the classroom while students are working on their solutions and class meeting time is focused on discussion.

Projects have been and are being used as a significant component in many different courses. We describe only the recent related work for sake of brevity. Krutz et al. [12] use real-world open source projects for teaching software testing. Szabo [19] describes using existing projects with real bugs for teaching software maintenance. Bloomfield et al. [2] describe a two-semester capstone project where students develop software for local non-profit organizations. Whereas the projects described in these related works are generally based on implementing a particular specification, our course and project structure (as described in Section 3) is different in that the problems are open-ended and the emphasis is on devising, analyzing, and communicating solutions, in addition to implementing them.

There has been a lot of work in using simulation-oriented games for teaching various aspects of CS. SimSE [16] uses a game-based simulation environment for teaching software process and software engineering. Wu's Castle [5] is a game to teach students basic programming constructs such as loops and arrays. Gibson and Bell [8] have conducted a large survey that evaluates games for teaching CS. Their survey (and the references therein) lists games on a variety of topics such as binary numbers, basic programming concepts, graph algorithms, and network protocols. All these papers focus on teaching a very specific CS aspect and are, by definition, narrow in scope. Our course is designed to do the opposite—it is intended as a broad upper level course that builds on courses such programming, data structures, algorithms, and software engineering and uses them as tools in solving problems.

Finally, soft skills such as communication and teamwork are becoming an increasingly important aspect for a Computer Scientist [6] and are found to be "unfortunately lacking in many new graduates" [11]. There has been some recent work that aims to integrate soft skills in various CS classes. Hoffman et al. [10] describe using "workplace scenarios" to teach communication skills in various classes. Hazzan and Har-Shai [9] draw the analogy between soft skills and soft computer science concepts such as abstraction and propose that both these can be learnt gradually by students over a period of time. Burns et al. [3] describe their efforts on teaching soft skills by having student teams partner with middle school teacher teams to help the latter integrate computing into their teaching and curriculum. Our course also aims to improve students' soft skills and is complementary to all of these efforts.

7. RECOMMENDATIONS

We conclude this paper with recommendations for others who might want to adopt this course.

7.1 Teaching the Class

The role of the instructor in this class is, paradoxically, to avoid instructing. As discussion of a problem progresses, the ideas should be coming from the students during (and between) class discussions. The role of the instructor is to structure the discussion so that progress can be made. Our advice on the instructor's role during class time is summarized below.

- Have a structure for class discussions. There should be an expectation that every student will be called on at some point, even when their hand is not raised. Discussions can be based upon results obtained since the previous class, with the focus moving from group to group. Or discussions can be open and broad, particularly in the first class for a problem cycle.
- Listen to a student present an idea, then highlight any insights and restate them in a parsimonious way. Sometimes ideas come up that could take the discussion in a whole new direction. If the previous direction of the discussion still has room for elaboration, put the new idea aside and come back to it later. It's easier to have discussions in a depth first rather than breadth first manner.
- If a student asks a question, pose the question to the class rather than trying to answer it.
- Avoid trying to solve the problems ahead of time. That way, the instructor won't inadvertently push the discussion in a particular direction. The most inspiring moments for an instructor happen when students come up with good ideas that the instructor would never have thought of.
- Be strict about participation. This means (a) eliminating side conversations and prohibiting the use of cell-phones and laptops during class discussions, and (b) limiting the number of allowed absences to a very small number.

Instructor preparation for a particular class hour is small, and the instructor will improvise as the discussion follows an unpredictable course. By improvising in this way, students get to see a more natural "way of thinking" about CS. The primary preparation effort for the class occurs before the class starts, when selecting a set of problems for the term.

7.2 What makes a good problem?

We recommend trying to choose different types of problems over the course of the semester if possible. Games are a good because they seem to get students motivated by friendly competition. If one chooses more than one game-type problem,

try to make them different (e.g., one two-player game and another many-player game). We suggest choosing problems that could have come from the real world, and are phrased within the context of simulating a (somewhat abstracted) real-world scenario. It's important to choose problems that have good visualizations, preferably interactive ones. A high quality interface can give the students direct feedback about the mechanics (and bugs) of their solutions, both in class and outside of class. Other important considerations include choosing/controlling the appropriate level of difficulty (using several parameters such as board size or number of players), whether the problem (or a close variant of it) has been solved before or extensively studied in the literature, and reusing problems used in the past (or from our repository) so that students can potentially have access to previous work done and build on it. Finally like Knuth, we aim to choose problems that are open-ended, and likely to be fun.

8. ACKNOWLEDGMENTS

The authors would like to thank their hard-working teaching assistants, without whom the courses would not have been nearly as successful.

9. REFERENCES

[1] B. S. Bloom. *Taxonomy of Educational Objectives: The Classification of Education Goals. Cognitive Domain. Handbook 1.* Longman, 1956.

[2] A. Bloomfield, M. Sherriff, and K. Williams. A service learning practicum capstone. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 265–270, New York, NY, USA, 2014. ACM.

[3] R. Burns, L. Pollock, and T. Harvey. Integrating hard and soft skills: Software engineers serving middle school teachers. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 209–214, New York, NY, USA, 2012. ACM.

[4] M. J. Clancy and D. E. Knuth. A Programming and Problem-Solving Seminar. Technical Report stan-cs-77-606, Computer Science Department, Stanford University, 1977. http://infolab.stanford.edu/pub/cstr/reports/cs/tr/77/606/CS-TR-77-606.pdf.

[5] M. Eagle and T. Barnes. Experimental evaluation of an educational game for improved learning in introductory computing. *SIGCSE Bull.*, 41:321–325, March 2009.

[6] H. A. Etlinger. A framework in which to teach (technical) communication to computer science majors. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 122–126, New York, NY, USA, 2006. ACM.

[7] E. F. Gehringer and B. W. Peddycord, III. The inverted-lecture model: A case study in computer architecture. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 489–494, New York, NY, USA, 2013. ACM.

[8] B. Gibson and T. Bell. Evaluation of games for teaching computer science. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, WiPSE '13, pages 51–60, New York, NY, USA, 2013. ACM.

[9] O. Hazzan and G. Har-Shai. Teaching computer science soft skills as soft concepts. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 59–64, New York, NY, USA, 2013. ACM.

[10] M. E. Hoffman, P. V. Anderson, and M. Gustafsson. Workplace scenarios to integrate communication skills and content: A case study. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 349–354, New York, NY, USA, 2014. ACM.

[11] J. L. Kayfetz and K. C. Almeroth. Creating innovative writing instruction for computer science graduate students. In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, pages T4F–1. IEEE, 2008.

[12] D. E. Krutz, S. A. Malachowsky, and T. Reichlmayr. Using a real world project in a software testing course. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 49–54, New York, NY, USA, 2014. ACM.

[13] B. L. Kurtz, J. B. Fenwick, R. Tashakkori, A. Esmail, and S. R. Tate. Active learning during lecture using tablets. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 121–126, New York, NY, USA, 2014. ACM.

[14] M. J. Lage, G. J. Platt, and M. Treglia. Inverting the classroom: A gateway to creating an inclusive learning environment. *The Journal of Economic Education*, 31(1):30–43, 2000.

[15] K. Lockwood and R. Esselstein. The inverted classroom and the cs curriculum. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 113–118, New York, NY, USA, 2013. ACM.

[16] E. O. Navarro and A. van der Hoek. SimSE: an educational simulation game for teaching the software engineering process. In *Proc. of the 9th annual SIGCSE conf. on Innovation and technology in CS education*, ITiCSE '04, pages 233–233, 2004.

[17] K. A. Ross and S. R. Shamoun. Programming and Problem Solving: A Transcript of the Spring 1999 Class. Technical Report cucs-018-99, Department of Computer Science, Columbia University, 1999. http://www.cs.columbia.edu/~library/TR-repository/reports/reports-1999/cucs-018-99.pdf.

[18] D. Shasha and C. Lazere. *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists.* Springer-Verlag, New York, August 1995.

[19] C. Szabo. Student projects are not throwaways: Teaching practical software maintenance in a software engineering course. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 55–60, New York, NY, USA, 2014. ACM.

[20] D. F. Wood. Problem based learning. *Bmj*, 326(7384):328–330, 2003.

[21] C. V. Wyk and D. E. Knuth. A Programming and Problem-Solving Seminar. Technical Report stan-cs-79-707, Computer Science Department, Stanford University, 1979. http://infolab.stanford.edu/pub/cstr/reports/cs/tr/79/707/CS-TR-79-707.pdf.