MOVING AWAY FROM PROGRAMMING AND TOWARDS COMPUTER SCIENCE IN THE CS FIRST YEAR

James Heliotis
Department of Computer Science
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623-5608
585-475-6133
jeh@cs.rit.edu

Richard Zanibbi
Department of Computer Science
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623-5608
585-475-5023
rlaz@cs.rit.edu

ABSTRACT

After completing a pilot study using the Python programming language to transition to Java within our first-year introductory programming sequence, our department opted to make a more radical change. We assert that our students are better served in their first year of study by a focus on problems in computer science and their solutions, rather than programming. Our new introductory sequence emphasizes algorithm development and analysis, object-oriented design, and testing. As in our pilot, programming is first done in Python, switching to Java when object-oriented design and static typing become advantageous. Students reported liking the problem focus of the courses, while the distribution of grades remained similar to those in previous years. As a result, our department will be discontinuing our earlier introductory sequence, and offering the new problem-based one to all the groups of students our department services beginning in Fall 2010.

1. INTRODUCTION

Our computer science curriculum has been grounded in object-oriented programming and design since the early 1990's. A significant change was made in 1998, when we switched from using Eiffel to Java as the programming language for our introductory programming sequence. At that time, there were clear advantages to using Java (e.g. platform independence), but over the last decade the language has grown grew to include inner classes, generics, the Swing GUI library, and many other features. This growing complexity has made it increasingly difficult to teach basic computing concepts and techniques. Others have reported similar concerns [2].

One can teach a restricted subset of Java, but this raises two issues within our curriculum. First, some Java features, most notably the purely class-based program structure and the strong typing, including generics, are unavoidable from the start. Second, our students are required to complete a year of co-op (industry internships), and it would be a disservice to teach languages our students would not likely use during their co-op placements.

Python has attracted attention in recent years as a good first language for instruction [5,9]. In an earlier pilot study [8], we used Python for the first four months of the first-year sequence and then switched to Java. The transition to Java proved quite smooth and uneventful, even with similar learning outcomes as for the original sequence. Benefits included Python being syntactically simpler, object-optional (e.g. not requiring class definitions to implement simple procedures), having an interpreter with a *read-eval-print* loop, and being dynamically typed (removing the need for type declarations). Other languages, such as JavaScript, PHP, Python, Ruby or Scheme [2] might be used to similar effect.

Programming language aside, a larger concern was that many of our faculty believed students were not receiving a solid grounding in computer science as a discipline. Our introductory courses' programming focus seemed to indicate that computer scientists were simply "code monkeys". We also came to feel that our students might better appreciate, and therefore understand, design concepts such as object-orientation if they first worked on large projects without them. A group of faculty members set out in early 2009 to design a course sequence that emphasized solving problems of interest to computer scientists. Originally a pure problem-based approach was considered [10,11], where groups of students solve hard open problems independently with instructor guidance. Based on reported experiences with pure problem-based learning [3,4,6,7], we chose to adopt a more structured approach.

The goals for our new introductory sequence include:

1. Emphasizing computational problem-solving [10, 11, 13], rather than programming, introducing language constructs only when needed to solve a problem
2. Identifying four core computational problem-solving tasks: (1) problem definition, (2) developing solutions, (3) implementing solutions, and (4) testing solutions, and organizing the presentation and study of problems explicitly around these tasks
3. Providing a problem-based course delivery (similar to a case-based approach), with students regularly practicing the four problem-solving tasks in lecture, lab, and homework assignments
4. Introducing recursive algorithms and functional programming early [2]
5. Having students regularly work in teams, communicate orally and in writing, and work on open-ended problems (through larger assignments, i.e. projects)

We compare the new and old course sequences further in Section 2, summarize our approach to problem-based course delivery in Section 3, and present student outcomes and feedback for our first offering of the new introductory sequence in the 2009-2010 academic year in Section 4.

2. COMPARISON OF OLD AND NEW INTRODUCTORY COMPUTER SCIENCE COURSE SEQUENCES

Our introductory computer science course sequence is comprised of three courses, as our university uses an academic calendar organized around quarters of 10 weeks duration.

Our previous Java-based curriculum was focused on helping students avoid poor programming habits and designs early, and preparing them for programming during their co-op positions. Students attended three hours of lecture per week, along with a two-hour lab supervised by a faculty member or a student lab instructor. Lab work was completed independently. In the second and third courses work was also assigned in the form of projects, 2 per course. Some of these were done in groups of 2 or 3 students. The first course introduced simple Java programming language constructs, including writing classes. The second course covered inheritance, and language libraries including collection classes, Swing (GUI), threads, and networking. In the third course, specific algorithms such as searching/sorting/hashing and data structures like trees and graphs were introduced. Starting in the first course, students were required to provide detailed comments (via *javadoc*) and use version control software (RCS) in the second course.

We keep programming style requirements and tools simple in the new sequence. Our students are expected to use only a text editor or the Python IDLE tool to create their programs in the first course, and require only short strings commenting the purpose of functions and any pre-conditions. Version control is not introduced until the third course.

Each week in the new introductory sequence, a class meets for a single two-hour lecture with the instructor, and then a two-hour lab. There is also a one-hour review session with an upper-class undergraduate student offered each week (a *supplemental instruction (SI) leader* [1]). Table 1 shows the sequence of topics covered in the new sequence each week during the first year.

Week	Problem-Based Intro to CS (CS 241)	Data Structures for Problem Solving (CS 242)	Object-Oriented Programming in Java (CS 243)
1	Introduction to CS	Linked list implementation	Java syntax (vs. Python)
2	Recursion	Stack & queue implementation	Classes
3	Advanced recursion	Backtracking	Interfaces: type inheritance
4	Strings, lists	Dynamic programming	Subclasses: implementation inheritance
5	(exam week; review)	(exam week; review)	OO design tips (+ midterm exam)
6	List operations, complexity analysis	Hashing	Exceptions, the Command pattern
7	Searching and sorting ($O(n^2)$)	Sorting (merge/quick)	The Observer pattern, Swing library
8	Trees	Dijkstra's shortest path algorithm	The Collection library, iterators
9	Graphs	Heap sort	Comparators, I/O streams
10	(review for final exam)	(review for final exam)	Factory, Template Method patterns

Table 1. Topics in our new introductory CS sequence, given over three quarters in one academic year

The first two courses in the new sequence, A Problem-Based Introduction to Computer Science and Data Structures for Problem Solving, are taught using Python. Although there is a push towards a functional style in the algorithms and implementations, imperative techniques are taught as well. Students learn how to use objects, but not design them, through the use of the standard Python library and a turtle graphics package used in many of the labs. The second course uses Python classes as simple records -- no methods (besides a constructor), but with public attributes -- as the building blocks of all of

the classic data structures. The third course is Object-Oriented Programming in Java. After a brief discussion of syntax differences, students are taught how to design and implement their own objects. A study of the design uses of interfaces and inheritance immediately follows. However, even here the lecture and lab material are motivated by problems where previous implementation techniques are shown to be awkward. These often fall into the topic of "brute force" programming versus extensibility, where widely used design patterns are shown to be of great help.

A small amount of the UML is presented informally in the third course, as a way to express object designs without code, following on the use of pseudo code to define algorithms in the first two courses. The third course continues with concepts such as exceptions, object-oriented frameworks for data structures and graphical user interfaces, in addition to common design patterns. The course textbook [12] focuses on effective design principles using Java for implementation. We expect our students to use the abundant on-line materials to look up Java language and library features.

3. PROBLEM-BASED COURSE DELIVERY

In the new sequence, lectures are delivered using a problem-based approach similar to case-based instruction. Concepts and language structures are introduced in the context of specific problems to solve. Initial rough solutions are sketched out with help from the students at the board, often before the concepts needed to solve the problem are presented. For example, in week 1 of the first course, students were presented with the problem of moving through a maze. The class discussed how to represent a maze geometrically, and then how to solve the maze. The instructor then introduced simple turtle graphics commands in Python for drawing a maze and moving a turtle through the maze, and implemented a solution for a maze live in-class. Notes for lecture problems are provided, using templates that explicitly identify our four core problem-solving tasks (definition, solution, implementation, and testing). Implementations of solutions to lecture problems, along with code developed in class are also made available. As problems become more complicated it is not always possible to consider all four problem-solving tasks during lecture, but instructors try to at least occasionally move from problem definition through testing a solution, to illustrate the complete problem-solving process.

In lab, a problem similar to those in lecture is given out on paper to the students. In week 1 of the first course, the lab required students to develop pseudo-code to write 'Hello World' using turtle graphics. Lecture sections, of size 30-60 students, are split into two sub-sections for lab. In the first hour of lab (the problem-solving session), the instructor presents a problem and the solution requirements (e.g. an algorithm in pseudo code and related test cases). Students are split into teams of 3-5. These teams produce solutions using only pen and paper. The instructor and two student assistants are available to answer questions from the groups. At the end of the lab, at least some of the groups present their solutions using a document projector, and the class discusses them. Students then go into the computer labs to start on their individual implementations of a solution, which may be different from those of their group or the solutions presented. Note that students were never asked to produce actual Python or Java code in the problem-solving portion of the labs.

Problems in the first two courses made use of turtle graphics, simple string, tree, and graph problems, and sorting and searching. Some examples include drawing recursive figures (lectures, labs and homework in the first course), a Sudoku solver (lab, second course), defining and searching decision trees (lab, first course), and implementing Dijkstra's shortest path algorithm (lecture and lab, second course). At the end of the first course, in lecture students learned about solving mazes represented as graphs through recursive depth-first searching, rather than simply drawing a solution. Examples from the third course include designing objects and methods to implement a car radio's user interface, developing a hierarchy of classes that describe popular family games, choosing a GUI layout for the "Nurikabe" puzzle, and using decorator classes to add encryption/decryption capability to the Java I/O class library.

3.1 Larger Problems: Projects in the Second and Third Courses

In the second and third courses, projects are assigned, with the intention that these be more open-ended problems. The project in course two is split into three parts that are done in the labs on certain weeks, allowing problem-solving sessions for the project as well. In the third course, two projects are assigned in parallel with the labs, making the pre-coding work for each project an out-of-class assignment, due several weeks before the code for the project must be delivered. The final code deliverables for the one of the two projects in the third course is individual; the other is optionally by two-person teams. The proportion of the grade assigned to the sole second course project was the same as the two in the third course combined.

The first project was completed in the second course. We used the Amazing Labyrinth maze game created by Ravensburger Toys, where multiple players try to reach series of treasures on the board and return to their home positions in order to win. Play is complicated by the fact that the maze changes after each player move. At each turn, a player must (1) insert the spare tile at one of the locations marked by arrows, shifting a row or column of the maze and popping a new tile out to be the spare, and (2) choose a reachable location on the board to move to (optionally staying in-place).

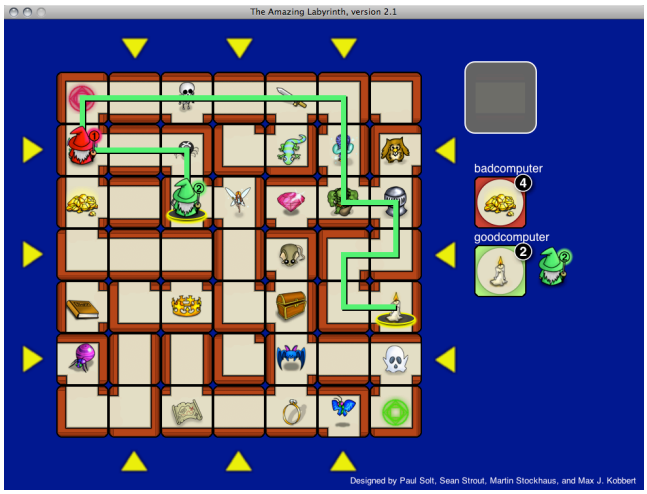

Figure 1: Amazing Labyrinth project (completed in the second course using Python).

The game engine provided to students displays the game (Figure 1 is a screenshot), and provides some error checking, e.g., to detect illegal moves. At each turn, the engine calls a player function, which must return an updated game state and the move to be made. Player modules maintain the game state, after initialization; the engine only records the game state for the player function. The game engine supports multiple players. At the end of the course there was a competition within and then between sections, with prizes awarded to the top teams (gift certificates at the college book store). Three labs in the second course are devoted to the project. Students have to provide pseudo code for (1) a breadth-first search over a static board, (2) returning a move and game state update when playing the game as a single player, and (3) returning a move and game state update when multiple opponents are involved. Solutions are not presented in the problem-solving session (lab) for the third component, to keep the end-of-quarter competition interesting.

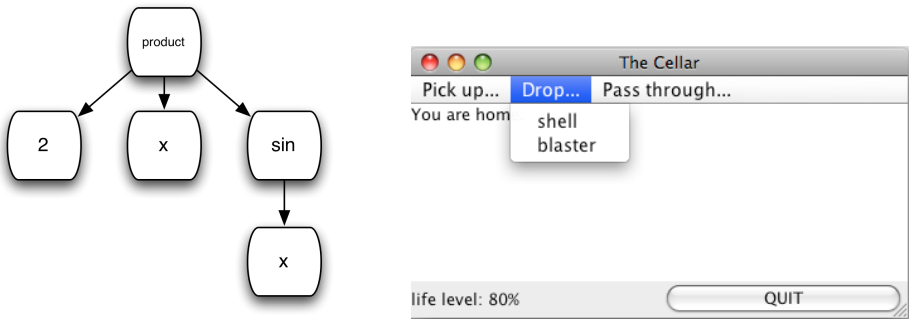

Figure 2: a) Functions project (Java) b) Crude mockup of The Cellar project (Java/Swing)

In the third course, projects are assigned in parallel with the labs. The workload is thus higher. The first project, on the theme of classes and interfaces, is to construct an operator tree representing a hierarchy of primitive, arithmetic, and trigonometric functions, as illustrated in Figure 2a. The functions, represented as primitive constants and variables, and composites of other functions, can be evaluated, differentiated, integrated, and plotted. This project mostly illustrates the usefulness of polymorphism by performing recursive operations on a heterogeneous tree. The students are not provided with any interfaces, just precise verbal descriptions and a simple test program. The first deliverable requires an object-oriented design in UML, a few of the easier classes implemented, and a better test program. The final deliverable requires a complete implementation, plus a revised design document and an enhanced test program.

The third project, emphasizing general OO design and use of the model-view-controller style, is an old-fashioned dungeon-crawling game inspired by the Rogue/NetHack genre of games. In this case, because the specification is more of a set of user requirements, the students have much more design freedom. Only a design is required for the first deliverable. It focuses on data structures, class relationships, and some dynamic object interactions. There is a graphical user interface. The entire implementation is submitted for the final deliverable. A possible GUI realization of the game is shown in Figure 2b, but students had complete freedom here as well.

4. STUDENT PERFORMANCE

In the first year of the offering, our classes were populated with 190 students from only two majors, Computer Science and Software Engineering. In 2010 we are adding all the other majors who normally take our freshman sequence (see section 4.2). Figures 3 and 4 show the grade results at the end of the second and third courses, namely Data Structures for Problem Solving and Object-Oriented Programming in Java. The graphs compare those grades with those students from the same majors who took the old CS2 and CS3 courses in the previous three years. There was little change in the grades from previous years; the proportions and pattern remained roughly the same. We are planning a more detailed analysis after the courses have been offered for a full two years.

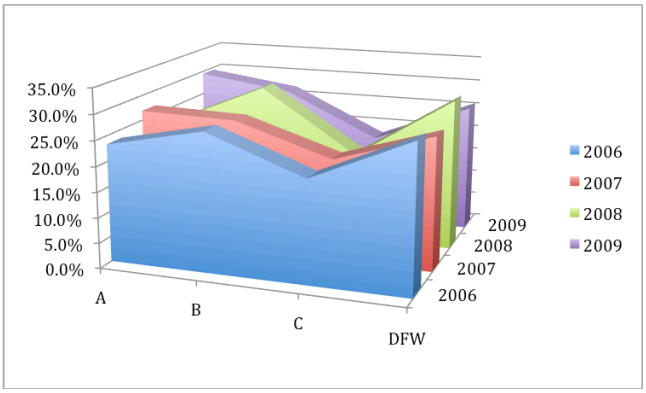

Figure 3: SE/CS grades in the second course

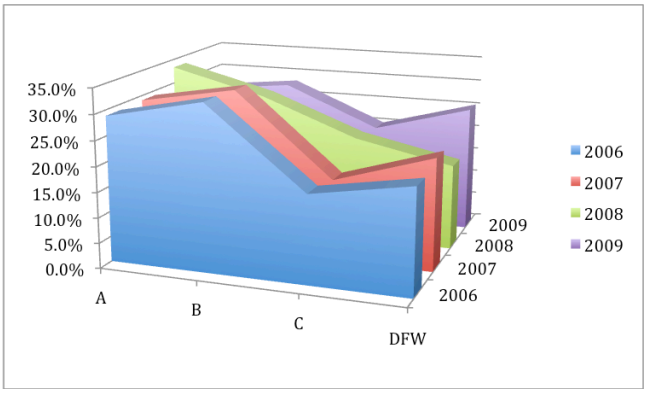

Figure 4: SE/CS grades in the third course

4.1 Questionnaire Responses

We requested that students complete a voluntary on-line survey after the end of the second course, the end of the period of the most radical change in our curriculum. Despite several reminders, response was low – under 17% (31 of 190 students). We provide a summary of responses on questions about the course in general below (additional questions asked about specific labs and the course project). Each of the following questions was answered using a five-point Likert scale (with values 1-5 labeled by: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree).

- Students indicated problem solving sessions (labs) were helpful: 54.84% 'Agree'; 22.58% 'Strongly Agree'
- Over half the respondents 'Agree' or 'Strongly Agree' that they 'appreciated the emphasis on computational problems rather than programming,' though approximately 30% were undecided.
- Over 93% of the respondents agreed or strongly agreed that two hours of lecture per week was sufficient.

Students were also asked about the course project, Amazing Labyrinth. A majority reported that the project was difficult (95.99%: 'Appropriate', 'Hard', or 'Very Hard'), but also was 'enjoyable'/'very enjoyable' (54.85%) to work on; 25.81% reported they felt 'neutral' about the project, while 19.35% reported the project was 'unpleasant'; no student surveyed indicated the project was 'very unpleasant.'

There was no similar questionnaire done for the other two courses during the 2009-10 school year.

4.2 Discussion

We first comment on what we view as encouraging outcomes for the new introductory sequence so far. First, we anticipate a reduction in D and F grades and withdrawals once instructors are not developing materials from scratch and students have access to upper-class students who have taken the same courses. Retention of students in the CS program changed little from previous years. We are hopeful that we will be able to increase retention next year.

From the survey, the students also seem to appreciate the new problem-based approach, particularly in the labs. Surprisingly, students frequently reported liking the use of pseudo code, and in some cases requested more time be spent on it in-class and lab. The students' familiarity with pseudo code made it much easier to teach algorithms for advanced concepts, e.g., heaps, later in the courses.

The project in the second course was for the most part well received, and students had a very good time at the final competition. This may have been a positive community-building activity as well, which is something that many have opined is lacking in traditional CS programs.

We must also admit to some disappointments.

In a small number of cases groups 'fell apart' in the process of completing the project in the second course. We feel this is independent of the structure of the course, as this is always a risk with teamwork.

Also, while we began teaching functional programming in week 2 of the sequence, we did not have time to teach functional programming as much as we had hoped. Students did get a solid introduction to inputs and outputs of functions, and thinking carefully about the design of individual functions. We also covered tail and non-tail recursion, but discussion of passing functions (closures) as arguments was not understood by many of our students.

About 44% of the survey respondents agreed or strongly agreed that more time should have been spent discussing programming in Python during lecture. If we maintain our desire to deemphasize programming in the lectures, this issue needs be addressed through some other kind of practice session. In 2010-11 we will be moving to a more standard recitation section run by carefully chosen teaching assistants. Although perhaps useful in a general way to the students, the SI model we used in the previous year severely restricts how much the professors and SI leaders can cooperate on teaching plans, In addition, SI is not just about course content, but about good study habits as well.

Overall, we remain enthusiastic about our progress and the new mood around the introductory sequence. After reviewing our results with them, we have received the permission of chairs of other departments who normally take the CS intro sequence to enroll their students in our new courses. Their programs include Computer Engineering, Bioinformatics, and Computational Mathematics among others. We are currently in the process of training more members of the faculty in the delivery style of the new courses for the 2010 fall quarter and will be collecting more data in the coming year.

ACKNOWLEDGEMENTS

Ivona Bezakova, Rajendra Raj, and Arthur Nunes-Harwitt helped produce the original design for the first-year courses. Sean Strout, Ivona Bezakova, Zachary Butler, Matthew Fluet, and both the authors developed the courses. Sean Strout and Paul Solt were the principal developers of our Amazing Labyrinth game engine.

REFERENCES

[1] Arendale, D., Supplemental instruction (SI): review of research concerning the effectiveness of SI from the University of Missouri – Kansas City and other institutions from across the United States, *Proceedings of the 17th and 18th Annual Institutes for Learning Assistance Professionals: 1996 and 1997*, 1-25, 1997.

[2] Bloch, S. A., Scheme and Java in the first year, *Journal of Computing Sciences in Colleges*. 15, (5), 157-165, 2000.

[3] Cavedon, L., Harland, J. Padgham, L., Problem-based learning with technological support in an AI subject: description and evaluation, *Proceedings of the 2nd Australian conference on Computer Science Education*, 191-200, 1997.

[4] Davis, T. A., Graphics-based learning in first-year computer science, *Computer Graphics Forum*, 26, (4), 747-742, 2007.

[5] Downey, A. B., Python as a first language: pre-conference workshop, *Journal of Computing Sciences in Small Colleges*, 22, (6), 3-4, 2007.

[6] García-Famoso, M., Problem-based learning: a case study in computer science, *Proceedings of the International Conference on Multimedia and Information and Communication Technologies in Education*, 2005.

[7] Hamalainen, W., Problem-based learning of theoretical computer science, *Proceedings of the 34th Annual Conference on Frontiers in Education*, 3, 20-23, 2004.

[8] Heliotis, J. E., Easing up on the introductory computer science syllabus: a shift from syntax to concepts, *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 2009.

[9] Ranum, D., Miller, B., Zelle, J., Guzdial, M., Successful approaches to teaching introductory computer science courses with Python, *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 38, (1), 396-397, 2006.

[10] Savin-Baden, M., Facilitating Problem-Based Learning, Berkshire, UK: McGraw-Hill, 2003.

[11] Savin-Baden, M., Major, C. H., Foundations of Problem-Based Learning, Berkshire, UK: McGraw-Hill, 2004.

[12] Skrien, D., Object-Oriented Design Using Java, New York, NY: McGraw-Hill, 2008.

[13] Wing, J., Computational Thinking, *Communications of the ACM*, 49, (3), 33-35, 2006.