EXPERIENCES INCORPORATING JAVA INTO THE INTRODUCTORY SEQUENCE*

Brad Richards
Computer Science Department
Vassar College
Poughkeepsie, NY 12604
richards@cs.vassar.edu

ABSTRACT

This paper describes a restructuring of our introductory sequence that resulted in the adoption of Java in our data structures course. Our motivation and plans are discussed, and our experiences - not always positive - are presented. While some of these experiences are specific to our functional-first introductory sequence, issues such as the transition from Java to C++ and the impact on later courses are likely to arise in most departments.

1. INTRODUCTION

Java caught the eye of Computer Science educators almost as soon as it was introduced, and its merits as an introductory programming language have been extolled in numerous papers [2,3,9,12,15]. But few papers have documented the decision process leading to the adoption of Java in a given department, and little mention has been made of its impact upon a curriculum as a whole. This paper reports on our experiences integrating Java into the introductory sequence. It describes the discussion within our department that led to the adoption of Java, the specifics of its use in our curriculum, and the sometimes unexpected impact on later courses.

2. BACKGROUND

Founded in 1861, Vassar College is a residential, coeducational liberal arts college with approximately 2400 students and 220 full-time faculty members. Vassar was a women's college until 1969, and the majority of the student body, 61%, still consists of women. Courses in computing have been offered since the 1960s (an IBM 360 arrived in 1967), and a Math/CS interdisciplinary degree was introduced in 1981. A Computer Science major was introduced in 1991 when CS emerged as a separate department. The department currently has six full-time faculty and about 50 majors. The 12 to 15 computer science majors that graduate each year split almost evenly between industry careers and graduate study.

In the terminology of the ACM/IEEE Curriculum 2001 report [13], our curriculum uses a "functional first" introductory approach with the rest of the program consisting of "topic-based" courses. As the department offers no courses targeted specifically at non-majors, CMPU 101 - "Problem Solving and Procedural Abstraction" has been designed both to introduce computing to students intending to major in Computer Science, and to give a broad introduction to the field to those who choose not to take additional courses in the department. This has influenced the design of the course in a number of ways. First, CMPU 101 introduces programming via Scheme, thereby limiting the time required to learn syntax and programming environments, and allowing a high-level presentation of problem solving, algorithms, and some basic data structures. Second, some historical background and "big ideas" in computing are included. Finally, in an effort to broaden the course, students are assigned readings from [16] covering topics such as ethics, copyright law, computational complexity, and the limits of computing. Students enrolling in CMPU 101 have a wide variety of backgrounds and interests. While not specifically required for any graduation requirement, the course is one way for students to satisfy a quantitative analysis requirement and is taken by approximately 100 students each year. Fewer than 15% typically continue on to major in Computer Science, with the remaining students split fairly evenly between majors in the arts and in the sciences.

Students who continue through the introductory sequence next take CMPU 102 - "Objects and Data Abstraction", a fairly traditional course on algorithms and data structures. The course is currently taught in Java, but used C++ until the recent curriculum change. CMPU 102 is tailored to take students from CMPU 101, with no imperative programming experience, and leverages off of the problem-solving and task decomposition skills students learn in CMPU 101. Both courses meet twice a week for 75 minutes, and include a two-hour closed lab session each week.

The third course in the introductory sequence is CMPU 203, a software design course. Titled "Software Development Methodology" until our recent curricular rearrangement, it introduced the Standard Template Library, techniques for object-oriented design, and gave students experience managing much more substantial programming projects. It has since been renamed "Data Structures and Software Systems" and its content has been modified somewhat. The revised versions of all three courses are described in Section 4.

3. MOTIVATION FOR CHANGE

In the spring of 2000, our department began deliberations over proposed modifications to our introductory sequence. The primary reason for change was our growing concern that there was insufficient time to cover the required material in each of the introductory courses. As is the case at many institutions, additional topics had been added over the years until it had become nearly impossible to cover them all in any reasonable amount of detail. We were also hoping to smooth the transition between CMPU 101 and 102. The Scheme-based 101 course was working very well, but the transition to C++ in 102 was difficult for most students, and required many weeks of course time that could otherwise have been spent introducing data structures and algorithms. Finally, we were intrigued by the possibility of adding Java to our curriculum in an effort to present object-oriented programming earlier and more easily, and discussed various ways in which it might serve our needs.

In our original curriculum, the amount of time devoted to learning C++ at the start of CMPU 102 meant that some common data structures (*e.g.* hash tables, heaps) had to be pushed back and covered in CMPU 203 instead. This was inconsistent with the stated goals of CMPU 203, a software design course, and made it difficult to cover the intended 203 material thoroughly. Our first decision was to accept reality and redesign CMPU 203 with the understanding that it needed to pick up the overflow from 102. As will be described in the next section, its content was partitioned and some of it moved to an upper-level elective course.

We then began considering changes to CMPU 101 and 102. The most vigorous discussions surrounded the use of Java. Our faculty were attracted to Java for reasons that are no doubt familiar to others. Chief among them in our case were the additional compiler and run-time checks, absence of explicit pointer syntax, uniformity of its object model, and opportunities for GUI applications. Its appeal to students could not be denied, and it was felt that the built-in threads and networking facilities could be useful for upper-level courses as well. But how best to integrate Java into our curriculum?

Some argued that Java was simple enough that we should teach CMPU 101 in Java. The next two courses in the sequence could then retain C++ and stay largely unchanged. As an additional benefit, students would already be comfortable with an object-oriented imperative language by the time they reached 102, allowing more time to be spent on algorithms and data structures in that course. However, the majority of the faculty felt that CMPU 101 should continue to be taught in Scheme. It had served us extremely well for over a decade, allowing us to quickly teach computing basics to both majors and non majors, levelling the playing field for students with absolutely no experience, and helping us to attract and retain women and minorities. (These observations have been documented elsewhere as well [5].)

4. THE NEW INTRODUCTORY SEQUENCE

In the end it was decided that CMPU 101 would remain essentially unchanged, though additional efforts would be made at the end of the semester to help prepare students to think imperatively. CMPU 102 would switch from C++ to Java, but still maintain its focus on presenting algorithms and data structures. Students would then learn C++ in the software

design course, CMPU 203, the third course in our introductory sequence. As mentioned above, some of the original content of CMPU 203 was removed to make room for additional material on data structures. The bulk of this material, having to do with more advanced software design issues, was moved to a 300-level course where it was felt that students could better appreciate it. (This is similar in spirit to the approach advocated in [6].)

In some ways the redesigned curriculum was an ambitious plan. Students would be forced to learn a new language in each of their first three courses. But we have long been in favor of having students learn a variety of languages, and students were already having to make the transition from Scheme to C++ - we suspected that using Java as a stepping stone between the two might actually make life easier for students, especially since the transition from Java to C++ was expected to be quick and painless.

The progression through the languages also seemed to make sense to us. Students would begin with a very high-level language, far removed from architectural details. Through the introduction of Java (via BlueJ [10] initially), they would begin to experience some lower-level mechanisms (*e.g.* destructive assignment, object creation, etc.), while still being shielded from details like explicit memory management. Finally, C++ would expose them to multiple inheritence, pointers, and templates.

5. EXPERIENCES

The Java-based version of CMPU 102 was taught for the first time in the fall of 2000, and we now have four semesters' experience with the new course. The modified CMPU 203 was also first taught in the fall of 2000, though it did not begin working with Java-trained students until the following semester. While a certain level of chaos and struggle is to be expected when implementing curricular change, the transition has been more troublesome than expected.

Some of the difficulties were simply the result of faculty needing to become more familiar with the content of the new courses, and were therefore to be expected. For example, faculty teaching CMPU 203 without having taught the new Java-based 102 knew less than in previous semesters about exactly what students had seen in 102. It is harder to effectively build off of previous material under such circumstances. And, as others have learned, it takes time to learn how best to teach an objects-first approach to computing.

But there were unexpected issues as well, and these cannot be so easily remedied. First, it is not obvious to us that the transition from Scheme to Java has been any easier for students than the transition from Scheme to C++ in the previous curriculum. We had assumed that students would take more quickly to the simpler language, but there is still a tremendous amount of syntax for students to learn before they can do any significant programming. Plus, as well as having to learn to think imperatively, they must now learn the concepts involved in object-oriented programming - something that was not immediately presented in the old C++-based version of 102. In short, our hopes of being able to cover additional algorithms and data structures in CMPU 102 have not been realized. We may experiment shortly with the use of Karel [1] or Jeroo [14] to help ease the transition.

More surprisingly to us, at least initially, was that students were finding the transition from Java to C++ quite difficult as well. We had naïvely assumed that this shift would be straightforward, if not trivial. After all, the syntax is quite similar, and they are both object-oriented languages. But even for the concepts that transfer directly from Java to C++, new syntax must often be learned. (Rushing through the barrage of syntax only serves to shake students' confidence, with perilous effects later in the course.) And the new topics introduced when learning C++ are significant: pointers and explicit memory management, a completely different I/O model, multiple inheritance, overloading, and the intricacies of the standard template library. It is a tremendous amount of new material, and requires many weeks of lecture to cover well. Most of this had been previously introduced in the C++-based version of CMPU 102, but must now be covered in 203. Thus, we are still not covering as much software design material as we had hoped in CMPU 203.

There is also anecdotal evidence suggesting that students are not as well prepared when they reach our intermediate-level classes as they had been under the old curriculum. For example, students in CMPU 241, our first theory and algorithms course, have never had more than one semester of experience with any given language, and seem to be less confident and competent programmers than those that were produced by the previous curriculum.

Finally, we are still discussing how our upper-level elective courses will need to change in light of the switch to Java in CMPU 102. Our department does not require a capstone project, but each of the upper-level courses contains a substantial implementation project. Some of these (*e.g.* networks, graphics) involve libraries of code provided by the instructor, and have been fine tuned over a number of years. Not surprisingly, faculty are reluctant to discard these projects or reimplement in Java. Supporting both Java and C++ and allowing students to make individual language choices in these courses will require a significant amount of work for faculty - both one-time reimplementation effort and an increase in the time required for ongoing maintenence and updates of the projects - and care must be taken that the difficulty of a project is similar across languages. On the other hand, if faculty require that students use only C++, it brings into question even more strongly the reaons for switching to Java in 102.

Discussions are already underway within the department about how best to remedy our current situation. No one has proposed returning to C++ in CMPU 102, but the majority are considering switching to Java in CMPU 101 as well. The hope is that by reducing the number of languages learned in the introductory sequence, more time will be available for academic content. Students would also have two semesters to familiarize themselves with Java and object-oriented programming before having to learn C++. But using Java in CMPU 101 is still an imperfect solution: It does not address the concerns regarding projects in upper-level courses, it creates a much more syntax-intensive first course, and would require additional course changes to ensure that majors are exposed to functional programming later in the curriculum.

6. SUMMARY

Our department began implementing a restructured curriculum in the fall of 2000. While the integration of Java was not our primary concern, it was one of the factors that motivated the changes to our introductory sequence. Our hope was that the use of Java in our second course (an introduction to algorithms and data structures) would smooth the transition from our Scheme-based first course, provide a better introduction to imperative and object-oriented programming concepts, and broaden our students' experiences by exposing them to an additional programming language. The third course in our introductory sequence would then introduce students to C++ and software design issues.

Unfortunately, the revised curriculum is not currently serving our needs as well as we had hoped and expected. The transition from Scheme to Java is still substantial and time consuming, and the move from Java to C++ seems to be nearly as difficult for students as the first transition. The time spent teaching language specifics is limiting the coverage of topics in both the Java and C++ courses, students seem generally less well prepared after the three-course introductory sequence than before, and the introduction of Java has forced us to reconsider the details of the rest of our curriculum.

Not all of the effects of the restructuring have been negative, however. By the end of our second course, students are more familiar with object-oriented programming than they were when it was taught in C++, and their experiences are broadened by the back-to-back exposure to Java and C++. Students also seem to enjoy the Java course more than they did when it was offered in C++. Still, departments expecting minimal impact from the integration of Java would be well advised to think carefully before acting.

ACKNOWLEDGEMENTS

The author thanks Susan Hert and the anonymous referees for their feedback on earlier drafts of this paper.

REFERENCES

[1] Byron Weber Becker. Teaching CS1 with karel the robot in java. In *Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education*, pages 50-54. ACM Press, 2001.

[2] Joseph Bergin. Java as a better C++. *ACM SIGPLAN Notices*, 31(11):21-27, 1996.

[3] Joseph Bergin, Thomas L. Naps, Constance G. Bland, Stephen J. Hartley, Mark A. Holliday, Pamela B. Lawhead, John Lewis, Myles F. McNally, Christopher H. Nevison, Cheng Ng, George J. Pothering, and Tommi Teräsvirta. Java resources for computer science instruction. In *Working Group reports of the 3rd annual SIGCSE/SIGCUE ITiCSE conference on Integrating technology into computer science education*, pages 14-34. ACM Press, 1998.

[4] Judith Bishop and Nigel Bishop. Object-orientation in java for scientific programmers. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 357-361. ACM Press, 2000.

[5] Stephen A. Block. Scheme and java in the first year. *The Journal of Computing in Small Colleges*, 15(5):157-165, 2000.

[6] Duane Buck and David J. Stucki. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 75-79. ACM Press, 2000.

[7] James Comer and Robert Roggio. Teaching a java-based CS1 course in an academically-diverse environment. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 142-146. ACM Press, 2002.

[8] Adair Dingle and Carol Zander. Assessing the ripple effect of cs1 language choice. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pages 85-93. The Consortium for Computing in Small Colleges, 2000.

[9] Jason Hong. The use of java as an introductory programming language. *Crossroads*, 4(4):8-13, 1998.

[10] Michael Kölling and John Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33-36. ACM Press, 2001.

[11] Elliot Koffman and Ursula Wolz. CS1 using java language features gently. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 40-43. ACM Press, 1999.

[12] Peter Martin. Java, the good, the bad and the ugly. *ACM SIGPLAN Notices*, 33(4):34-39, 1998.

[13] The Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal of Educational Resources in Computing (JERIC)*, 1(3es):1, 2001.

[14] Dean Sanders and Brian Dorn. Jeroo: a tool for introducing object-oriented programming. In *Proceedings of the 34th technical symposium on Computer science education*, pages 201-204. ACM Press, 2003.

[15] Paul Tyma. Why are we using java again? *Communications of the ACM*, 41(6):38-42, 1998.

[16] Henry M. Walker. *The Limits of Computing*. Jones and Bartlett, 1994.

[17] Mark Allen Weiss. Experiences teaching data structures with java. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 164-168. ACM Press, 1997.