New Methods for Teaching Programming Languages to both Engineering and Computer Science Students

Dr. Peter C. Patton, Professor
QMCS Department
St Thomas University, St. Paul, MN
pcpatton@stthomas.edu

Abstract

The author has been teaching programming and computer architecture since 1957 and has recently dramatically changed his approach with good results. The basic idea is first to focus or specialize programming course sections for student interest groups. For example, we have an Introductory Java course for science and engineering students, which employs classroom examples, exercises, and teamwork labs that deal with math, engineering, and science applications and coordinate these with the classes they are taking in those majors. We have designed and are now teaching an Advanced Java course that will be focused on preparation for the SUN Microsystems Java Developer Certification test. The approach, while new in our Computer Science curriculum will be modeled after current efforts to prepare Math Seniors to take the Actuarial Exam, or Engineering Seniors to take the Engineer-in-Training exam.

The second change in methodology is to first teach student at both levels new material by first *reading* programs rather than *writing* them. For years we have been teaching students a bit of syntax and then asking them to write a program, but even those who are able to learn by this "sink of swim" approach rarely end up able to read a program they have not written. We ask students to start each exercise with a blank piece of paper (or PDE screen) but there is nothing in real-world programming practice like that. A programmer, whether novice or experienced, usually begins with a working program or a piece of one and modifies or augments it to get the desired or specified result. Moreover, in many cases that prototype program has been automatically generated and may employ techniques unfamiliar to the developer. The new approach described here was developed by the author to use the same methods for teaching an artificial language one would use to teach a natural language.

Introduction

In the Quantitative Methods and Computer Science Department at St. Thomas University we have chose Java as the primary medium of programming language instruction. While most of our students are taking programming as a major or minor subject, we also have sections in Introductory Java each term as service courses for Science, Engineering, Math, Business, and Education majors. Although the same textbook is used, the lab sessions focus on problems drawn from student's major disciplines in an effort to overlap their labs and class projects in their major classes. Business application development in COBOL is also taught as a service course for the Business School and for Computer Science students who wish to learn a language besides Java. There is now increasing

demand to teach the C language for electrical engineering majors since it is holding on as a language for programming embedded computers in spite of increasing interest in MicroJava.

Teaching Programming Languages

When I graduated from college with a degree in engineering and applied physics in 1957 I took a job in the structures programming group at The Boeing Airplane Company in my hometown, Wichita, Kansas. It turned out that at this early date in the computer revolution I was the only person in the group that had studied computing in school, so in spite of my youth, they immediately put me in charge of training. When I wasn't teaching classes on programming or numerical analysis I was working in the systems group developing new programming languages. This was before compiler languages had become widely used so the languages we employed were interpretative languages, like Mandy Grems' BACAIC (Boeing Airplane Company Algebraic Interpretive Compiler), SPUR, a Boeing interpreter for the IBM 650, as well as specialized Floating Point and Double Precision Floating Point interpretive languages, Matrix interpretative languages, and so on. When we replaced the IBM 701 with a 709, we started using Fortran but the powerful matrix interpretative languages continued in use for some years. Like many industrial firms at that time, efforts were made to retrain draftsmen, time and motion study people, engineers of various kinds, account clerks, etc. to supplement the shortage of programmers. The IBM programmer capability test predicted that people with musical ability would make good programmers and this seemed to be the case, however my experience at the time indicated the chemical engineers always excelled. Perhaps it was their pilot plant training that made computing technology especially accessible to them, i.e., first building and testing a model of a process before committing to it. It seemed to me that everyone who really wanted to learn to program was able to do it and those that were not so strongly motivated were not. As computers have permeated technology and our technology-based society generally more and more people need to know how to program to at least some degree, not just the "heat-seekers."

Twenty years ago many students showed up as college freshmen with some knowledge of Basic language programming, often learned in junior high school. However, while today I occasionally find a student who has taken a C++ course in high school, most have no knowledge of programming at all. My current Introductory Java class of twenty students are all totally innocent of computer programming. This is not to say that they do not know how to use a computer; they type papers in Word, use Excel in accounting classes, prepare classroom presentation in PowerPoint, and of course many are skilled computer game players. There is an interesting difference between teaching C, C++, Java and the older mainframe procedural languages like Fortran, COBOL, and Algol. How could McCracken teach Algol in 100 pages, or Fortran in some 140, or COBOL, which had a lot of detail, in less than 200, and yet the Sun **Java Tutorial** series is three 950 page volumes and continually refers to the online Sun JavaDoc files?

I think there are two reasons, one of which is a consequence of the other. Today's OOP languages are rather abstract and procedural languages were relatively concrete;

consequently it takes a lot more working examples to communicate the former than it did the latter. In addition, the users of early programming languages had problems they wanted to solve on a computer; today's students want to learn *how to program a computer*, not just how to solve a given problem or problems. This is a subtle but significant difference in learner motivation.

Reading Before Writing

The instructional approach is simple. For the past two years I have been teaching introductory Java and COBOL classes to students by first teaching them to read the language before teaching them to write programs in it. The approach seems to encourage beginners and promotes classroom involvement and retention. It certainly avoids the panic of looking at a blank piece of paper (or today, a Notepad or PDE screen). I call this panic the Red Barber syndrome after the famous baseball sportswriter/sportscaster. He used to tell the story on himself of the twelve-year-old lad who approached him leaving a game and asked: "Gee, Mr. Barber, I want to be a sports writer just like you when I grow up; is it hard work?" To which Red replied: "No kid, there ain't nothing to it. You just go to one or two baseball games every day during the season, take notes, eat hot dogs and drink Coke, and then go back to your office, sit down at your desk, put a blank piece of paper in your typewriter, and, and, and … stare at it until drops of blood appear on your forehead."

Moreover, there is very little in professional programming practice that resembles starting with a blank screen for each job assignment or "new" program. Most programmers, and certainly most novice programmers, are usually given a working program to either document or change to make it do something additional, something differently, or something else altogether. Our approach begins each class with lectures out of the textbook, but paying special attention to explaining how the example programs work. In addition we bring them up in the TextPad PDE, execute them and then ask the class what changes they would like to see made. Sometimes introductory programming students can be very shy (for fear there is a computer heat-seeker or guru in the classroom), so we usually have a short list of proven mods to try out. The instructor making errors in front of the class while doing these mods is *not* a problem, in fact it enhances the reality of the computer programming learning experience for the students.

After about two weeks, or four lecture periods of 65 minutes and three labs of 100 minutes (which are used for demonstrations at first), we assign an in-class problem that is a working program. For this first lab assignment we ask the students to key in and run. The lab exercise is (a) to write a comment on each line telling what that line is doing, and to write a JavaDoc style summary of what the program as a whole is doing. A popular first exercise is to print out the first 20 Fibonacci numbers. Most the fourth lab session is spent dealing with the program development environment and explaining why the world's newest and most sophisticated language and the sophisticated three window CAD development environment put out their results in a DOS window! Students always seem to be disappointed at this. The second part of the lab for the more aggressive students is to printout the first 40 Fibonacci numbers. This is of course a trick problem,

since the numbers get larger and larger with each addition so the second twenty will not fit on a second line. About half the class is stunned by this development and doesn't get past it in the first lab session. Some become a bit more creative by making lemonade out of this lemon, This term I had three of the 20 students figure out how to printout the series as a single column, arguing that since each number was the sum of the two above it, that was a better presentation of the results. And, there is usually at least one true heat-seeker in each class; in this class that student printed out the first 40 numbers neatly arrayed in five columns. He had figured out how to use the \t or tab string-break command.

The Progression

The progression of lab exercises leads the student through the syntax of the language in such a way that he or she sees recommended use of each language feature and its effects before being asked to employ it. The exercises are taken from the textbook, which in this case is **Java Software Solutions** Third Edition, by John Lewis, and William Loftus (Boston: Addison-Wesley, 2003). Since the Introductory Java section I teach is entitled *Development of Scientific and Engineering Applications in Java*, many simple applications from mechanical and electrical engineering are used as supplementary classroom and laboratory examples as well. The progression of labs goes as follows:

1. Document the Fibonacci number calculation for the first 20 numbers in the series.
2. Extend the Fibonacci calculation to print the first 40 numbers based on a textbook model.
3. A series of labs involving modifying the drivers as above (2), followed by a series requiring the substitution of new methods in already well-understood drivers.
4. A documentation exercise on a relatively advanced instructor furnished program that illustrates most of the methods in the Java Array class.
5. Modifying a Sieve of Eratosthenes program that involves parallel one-dimensional arrays.
6. A lab involving sorting arrays.
7. A series involving the use of two-dimensional arrays to do matrix calculations.
8. A documentation exercise on a program to do the Gauss-Seidel method for solving n-linear equations in n-unknowns.
9. The textbook has a nice example of a Java program to solve the quadratic equation. We lead the class through the creation of an abstract data type ComplexDouble and a set of methods to do complex double precision arithmetic.
10. At this point we turn them loose to extend the textbook program to solve quadratic equations having complex roots.
11. Those (about half of the class) who finish this lab early are put into two or three person groups to convert their application to a Java applet. This seems to be the most popular lab exercise in the series. The half of the class that doesn't get that far in class begs to be allowed do this step in groups as "homework." This request is always granted.
12. At this point we encourage the class to form small groups, and turn them loose on clearly specified lab exercises to augment certain textbook programs with both

overloaded and overidden methods. In each case they begin with a working program.

13. The first "blank screen" exercise is the time-honored C++ textbook-programming example (but in Java, of course), i.e., a program to create 52 objects as a deck of cards and write a method to deal a hand of three cards. The two or three teams in the class who are able to finish this early are encouraged to write a driver to deal four bridge hands, and/or write a pseudo code for a driver to use their new class and its methods to play some card game they are familiar with.
14. The last lab is on recursive programming and employs the time-honored Ackerman Function, a simple but non-primitive recursive function with two integer arguments. They begin with a working program but are challenged to turn it into a non-recursive program that computes Ackerman's Function iteratively and to compare the efficiency of the two approaches.

How Do We Test?

The first mid-term exam is almost all *about Java,* i.e., they are asked question about the language and its syntax, asked to write down what furnished Java snippets will output, to modify a few simple methods, and to rewrite for loops as while loops, as do loops, etc., and to document a program using Uniform Modeling Language (UML), but finally to write two methods, one to sum 100 integers, and a boolean method named floatEquals that accepts three floating point numbers and tests to see if the first two are equal within the tolerance of the third. Only about half of the class can do this reliably at this point.

The second mid-term exam is a conventional *write a program to do this* sort of exam and the students typically are as good or better than the traditional sections with a blank screen by this point. The final exam is a two-hour test that qualifies the student to take the departments Advanced Java class. We do not have experience on whether these student do better in Advanced Java because most of them are taking Introductory Java to meet a science, engineering, or mathematics major requirement and do not go further. We also require a final project in addition to the final exam and encourage the students to write a Java program that supports a lab project in a major class.

Performance and grading are competency-based and everyone achieving a certain competency level get an A, a second level a B, the third level a C, etc. The grades run high with half or more of the class getting As, most of the others getting Bs and one or two Cs. I cannot answer the charge that since my students are science, engineering, and math majors, they are often considered to be more disciplined and career-directed than other students, then of course they do well and would probably do well no matter what I did in the classroom. My answer to this charge is that while it is true they are science and engineering majors they are *freshman* and many of them do change majors in their sophomore year. Of course, I can claim only two years experience with this teaching approach, and sighting only two robins doesn't declare it Spring. In two years, however we have used it on two Introductory Java sections, three Introductory COBOL sections and are currently using it in an Advanced Java section.

Does it Work?

I claim that students who learn Java this way learn to program faster and with less frustration than thumbing through a textbook trying to figure out arcane syntax and run-time error diagnostics. The QMCS department at St. Thomas University asks each instructor to turn in copies of the penultimate lab program inn each section to be evaluated by an instructor who is not teaching a Java section in the current term. Classes learning this way have excelled in these evaluations. To me however, a more important measure is the dropout rate. In these read before writing section the dropout rate has been one in twenty both years compared to much higher dropout rates in other classes. At the University of Minnesota years ago, I helped Professors Cecil Woods of the German Department and Russell Burris from Educational Psychology with a computer-based introductory German (syntax) course. In defending this approach before the House Higher Education Appropriations Committee I had to admit that while the examination results did not show the students from the computer-based sections to do significantly better statistically in the final exam, our experience over five years was that the traditional 20% dropout rate in beginning language courses went to none or at most 1 out of 20 in these sections. We are seeing the same result in Introductory Java.

The method works as well in introductory COBOL classes for both computer science students who have already taken Java and for business majors who have no programming experience at all. Naturally we teach COBOL as a dialect of the English language, which it was designed to be. Our original motivation for this was the assumption that no (or at least very few) new COBOL problems are being written today but that many millions of lines of COBOL are being maintained and/or being converted to Java. Perhaps 80% or more of the enterprise computing programs running today are written in COBOL and many of those (e.g., Hogan banking software) will be running 20 years hence. In this class we can give the students significant example programs to analyze, document, modify and develop skill as a *reader* of COBOL before they begin to write COBOL programs.

My Advanced Java section this term is an evening mostly adult class of mostly professional programmers or engineers who need to learn Java. They are all interested in sitting for the SUN Microsystems Java Programmer Certification exam after the class and we are supporting this goal with a reading before writing approach that seems to be particularly suited to the exam, which presents code snippets and asks the taker: “Will this code compile? If not, why not? If it compiles will it run to completion? If it does, what will the program print out?” All having multiple-choice answers, of course. The success of the teaching approach will depend on how many of the class of 16 are emboldened to sit for the exam, and how many of those who do so, actually pass it on the first try.