SCHEME AND JAVA IN THE FIRST YEAR

Stephen A. Bloch
Department of Math & Computer Science
Adelphi University
Garden City, NY 11530
516-877-4483
sbloch@adelphi.edu

ABSTRACT

We compare the experiences of several years of teaching beginning programming in Pascal, in Java, and in Scheme-followed-by-Java. In addition, we discuss the integration of PSP and "pair programming" into the first-year programming courses.

1 HISTORY

In 1997, the Adelphi University Department of Mathematics and Computer Science decided to re-examine its choices of programming languages. Our first-year courses, a fairly standard ACM CS1-CS2 sequence [ACM91], had been taught in Pascal for fifteen years, and students were exposed briefly to C and Scheme in a third-semester "Survey of Programming Languages" course. This half-semester was students' only exposure to the functional programming paradigm; the logical and object-oriented paradigms had no official place in the curriculum, although the author had been slipping some OOP into the fourth-semester "Principles of Programming Languages" course.

A number of options presented themselves. The College Entrance Exam Board had just approved C++ for the Advanced Placement test, so if we started in C++, we could both accept students with AP credit and prepare high school teachers to teach C++. In addition, C++ has relatively easy primitives for basic I/O. C had a simpler, more consistent syntax, and could be used as a stepping-stone to C++. Various extensions of Pascal were object-oriented, had fairly readable syntax, and would require less retraining for existing faculty. Eiffel offered a principled, contract-based approach to programming. Scheme had a simple, consistent syntax. SmallTalk had a simple, consistent syntax and was object-oriented. Java was object-oriented, had standardized graphics and networking libraries, and was "hot".

Each language also had its disadvantages. C++ had an enormous, inconsistent syntax, with dozens of obscure ways to insert bugs in any given program. The Java language and libraries seemed to change every few minutes. C and anything resembling Pascal had an "old-fashioned" cachet. Eiffel, Scheme, and SmallTalk were anything but "hot", and would be difficult to sell to the mothers of incoming freshmen (an important consideration, since Adelphi was in an enrollment crisis at the time).

At this stage the author attended CCSCNE 1998, including numerous talks on how to use C++ and/or Java in the first year. Speakers debated when to introduce objects and classes, when to introduce inheritance, and even when to introduce numbers. Several stated that OOP was an "advanced" technique, to be taught only after students were comfortable with the "simpler" procedural programming paradigm. The author disagreed, suggesting that OOP seemed "advanced" only to those of us who encountered procedural programming first, and that if it were taught on its own merits from the beginning, students could grasp it as a natural way to model the real world. I decided to teach the course in Java, introducing objects, classes, and inheritance as early as possible.

2 THE FIRST YEAR IN JAVA

In 1998-1999, I taught CS1 and CS2 entirely in Java, using a "lecture-in-a-lab" presentation mode with no teaching assistants. Students were enthusiastic about learning Java, and initial course enrollment approximately doubled from previous years. After a few weeks, of course, students realized they had to work, and several dropped the course at this point, as expected. More surprisingly, though, after six weeks the *majority* of students felt that they were "way behind the class, with no chance to catch up." On closer investigation, I realized that many had never mastered the basic development-environment skills necessary to create, compile, and run the simplest program. I postponed my schedule of lecture topics for a week to review "how to use the development environment," then for another week to review the most basic Java syntax (which they should have been practicing a month before). The year went on that way: every time I wanted to introduce an interesting new principle of programming, I had to spend precious lecture time on language syntax, deciphering cryptic error messages, and using the development environment. Students were dropping the course rapidly, and my own enthusiasm was waning.

Nonetheless, we did manage to cover the notions of object, class, inheritance, and class design, all of which had previously appeared only in the fourth semester. Conditionals, loops, and arrays were introduced months later than they had been in Pascal, but linked lists and recursion (using polymorphism in lieu of conditionals) were introduced early, with reasonable success. A few weeks of the second semester were spent on graphics and GUI programming, the reason the students wanted to learn Java in the first place; these topics had previously appeared only in a junior-level elective course. Overall, the pedagogical results seemed somewhat better than those of previous years in Pascal. (This conclusion is borne out by the professor teaching those students in several sophomore courses in Fall 1999.)

But some disturbing problems remained. Of the 49 students attending the first week of the first semester, 36 passed the first semester, 18 started the second semester, and 16 passed the second semester. Moreover, female students dropped out disproportionately: although Adelphi is about 60% female, the fraction of female students in the class started at 31% and dropped to 28% passing the first semester, 22% starting the second semester, and 18% passing the second semester. Some of this could be predicted, of course: few students other than CS majors and minors ever intended to take more than one semester, and it is no news that computer science majors are disproportionately male. But the retention rate, and its correlation with gender, are still lamentable. I speculate that it can be blamed in part on a "hacker mentality" in which the willingness to stay up later than one s classmates, tirelessly seeking and destroying error messages, seems more valuable than the abilities to plan creatively and collaborate constructively (for related speculations, see [Bar99]).

The experience of this course led me to several conclusions:

1. Both the students and the instructor spent far too much time on how to use the development platform and where to put semicolons and curly braces, leaving less time than desired for principles of program design and software engineering.
2. Many students were left with an impression of CS as the tedious, frustrating memorization of unmotivated rules, and the equally tedious and frustrating Quest for a Clean Compile (leaving the Working Program as an almost unattainable afterthought).
3. While OOP may not be too "advanced" for beginning programmers, Java may be, because it has so many syntax rules and requires so much overhead code for a typical CS1 program of a few dozen to a few hundred lines.
4. Most available Java development platforms are designed for professional programmers, not beginners. They have error messages that make no sense to beginners, require the user to set many options to create even the simplest programs, and allow the beginner to accidentally invoke advanced features of the language. (For example, a misplaced curly brace in Java can create an "inner class", which CS1 students don't need, and produce a remarkably misleading series of error messages.)
5. Most students have a hard time analyzing error: years of right/wrong grading in K-12 school have taught them to distinguish between "it works" and "it doesn't work", but they have trouble breaking down the latter category further, and their approach to debugging is random.
6 Most students, especially college freshmen, have difficulty managing their time and starting projects early enough to meet deadlines.
7. Students have difficulty distinguishing between accidental issues of platform and language syntax and fundamental issues of program design and software engineering.

3 THE TEACHSCHEME WORKSHOPS

In Summer, 1998, the author attended a workshop at Rice University on the "TeachScheme! Project" [Fel], a package of software, course materials, and pedagogical techniques based on Rice s own first-year program, which uses Scheme as the first language and Java as the second. The workshop was inspiring, and indeed I briefly considered spending the first few weeks of the course in Scheme, but concluded that the disadvantages of switching languages in the middle of the first semester outweighed the advantages of starting with a syntactically-simple language.

Most of the workshop participants were high-school teachers frustrated with the C++-based AP CS curriculum, although a few other college professors have participated too. The central principle of the TeachScheme! project is

Principle 1 *The first CS course is not about learning a programming language, it's about learning to solve problems algorithmically.*

From this perspective, *any* programming language is a necessary evil: necessary in order to write real programs that really work on real computers, but evil insofar as it distracts students from thinking about problem-solving.

Corollary 2 *Introduce only those language constructs that are necessary to teach programming principles.*

This principle is nothing new   I've heard it espoused by other speakers at CCSCNE. The next corollary is less widely used:

Corollary 3 *Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time.*

The "few language constructs" principle weighs against Java, and even more strongly against C++ with its bewildering morass of special syntax. The "one at a time" principle weighs against C++, and even more strongly against Java, in which dozens of keywords and syntactic features must be used by rote in even a "hello, world" program. The Scheme language, however, satisfies both requirements.

The TeachScheme! project comprises three pieces:

1. DrScheme, an interactive Scheme development environment intended for pedagogical use. Although it implements the full Scheme language with extensions for graphics, OOP, CGI, XML, etc, an emphasis has been placed on good error messages, helping users with syntax, indentation and parenthesis-matching, and a clean interface with a minimum of options to set. A notable feature is the provision of instructor-customizable *language levels* Beginner Student, Intermediate Student, Advanced Student, and Standard Scheme that treat common beginner mistakes as syntax errors rather than invocations of advanced language features. In addition, a "Stepper" feature allows users to single-step through the evaluation of an expression, watching the substitution of values for variables and sub-expressions. This helps not only to debug programs and to illustrate recursion but to concretize basic algebra for students who never quite "got it" in high school. Indeed, I know several middle- and high-school math teachers who use DrScheme successfully in their algebra classes.
2. A pedagogical approach based on "design recipes" (aka patterns), which help students to write a useful skeleton for a new function and avoid blank-page syndrome. The recipes depend heavily on students understanding of input and output data types: for example, if the input is a composite type with three variants, the program body will almost certainly be a conditional with three cases. The recipes also emphasize choosing good test cases for a program *before* coding it.
3. A textbook, *How to Design Programs* [FFFK99], which introduces the various design recipes together with numerous data structures and principles of program design.

The software, documentation, and a pre-release version of the textbook are available for free download on the Web [Fel]. The design recipes are applicable to programming in any language indeed, I translated many of them into Java for my 1998-1999 courses but the on-line materials present them in a Scheme context.

In Summer, 1999, I attended a followup workshop at Rice on how to introduce Java to students who have already had a TeachScheme!-based beginning programming course. I realized that many of Java s concepts could be introduced much more easily in a *second* course than a *first,* and I decided to follow the Rice model more closely this year: a first semester in Scheme, with the second semester in Java (using Rice s DrJava and/or Monash University s BlueJ [Blu], two beginner-friendly platforms in which users can evaluate individual expressions without writing a program around them a great help in testing).

4 PSP AND PAIR PROGRAMMING

I decided to weave several other threads into the 1999-2000 course: the Personal Software Process (PSP) [Bei99, HT99, Hum98, Wil97] and "Pair Programming" [BC99, WK99].

PSP, familiar to many CCSC participants, focuses not on designing data structures or programs but rather on recording one s use of time, the number of defects and the size of one s programs, analyzing these data, and using the results to estimate the size, number of defects, and time demands of subsequent projects. As I described it to my students on the first day, "you ll experiment on your own minds, so you can predict how they ll work in the future." The method promises not only improved skills in programming and in early estimation of project difficulty, but in time management in general (not confined to computer science; see [Bei99]).

"Pair Programming" is the practice of students writing programs as a team of two not by breaking up the problem for separate solution, but by working *together, at one workstation* on the *whole* program. Industrial experience has shown this technique to dramatically improve productivity and reduce defects [BC99]. It is still unknown how well it works in a beginning programming class, but several benefits are hypothesized [WK99]:

_ Faster discovery of errors through the "two pairs of eyes" phenomenon. As Weinberg writes, "The human eye has an almost infinite capacity for not seeing what it does not want to see.... Programmers, if left to their own devices, will ignore the most glaring errors in their output errors that anyone else can see in an instant." [Wei71] (quoted in [Boo94]);

_ Closer adherence to programming methodology as partners "keep one another honest";

_ Greater perspective by exposing each student to the programming styles of several partners over time;

_ Early experience at programming as *not* a solo affair (as it almost never is in industry);

_ Improved record-keeping (*e.g.* for the PSP approach described above): one student can record errors and their causes as they are found while the other does the actual typing to fix them;

_ More efficient use of lab time, as students help with one another s weaknesses rather than waiting helplessly for the instructor to get around to them;

_ Improved retention of female students by making programming a social, collaborative activity [Bar991].

The technique has pitfalls, of course. To prevent the weaker member of a team from becoming uninvolved, [BC99, WK99] both recommend that the less-experienced programmer do all the typing. To assign valid individual grades (and to give students a greater breadth of experience), I require students to switch partners between each of 8-10 programming exercises.

5 THE EXPERIENCE OF 1999-2000

As I write this final version of the paper, the second (Java) semester has just started. Although I expect to have more quantitative conclusions on retention, and qualitative conclusions on the Scheme-to-Java transition, by the time of the conference, several results are already evident:

1. Students had much less trouble with the DrScheme development environment than with CodeWarrior last year.
2. Students had much less trouble with Scheme language syntax than with Java syntax at the same time last year.
3. Much more class time was spent on principles of program design, and much less on syntax and platform, than in the same semester last year.
4. Students resist the record-keeping required for PSP, and prefer to copy down error messages than analyze their causes, but those who do the record-keeping are developing solid, consistent data on their programming productivity.

5. Students resist switching partners. Students also resist working together on *all parts* of a programming exercise, believing (perhaps incorrectly) that it s less work for each to do half of the exercise. In addition, since Adelphi is largely a commuter school, students claim to have difficulty meeting outside class.

Many topics have been introduced both earlier and more successfully than last year:

- Linked lists were introduced a week earlier than last year.
- Recursion was introduced a week earlier (and more successfully, based on students in-class questions and homework performance) than last year.
- Compound data ("structures" or "classes" with instance variables) were introduced two weeks earlier than last year, even though last year was explicitly object-oriented.
- Conditionals were introduced two months earlier than last year.
- Function composition was discussed two months earlier than last year.
- Choosing good test cases was discussed explicitly two months earlier than last year.
- Sorting was introduced four months earlier than last year.
- Binary trees were introduced five months earlier than last year.
- Graphics was introduced five months earlier than last year, even though last year was in Java.

In exchange, some things have been delayed:

- I/O programming was omitted completely from the first semester. In the author’s experience, I/O programming is tedious and dependent on language, operating system, and platform exactly what does *not* belong in CS1.
- Inheritance was removed from the first semester, although students were exposed to the idea of types and sub-types.
- Strings were mentioned, but de-emphasized in favor of symbols, in the first semester.
- Class-based polymorphism was removed from the first semester, but students started writing "functions that work differently on different-type inputs," and explicitly looking for this pattern, in the first month.
- Assignment statements and other "state"-based programming techniques were omitted completely from the first semester. These techniques have a legitimate place in Scheme programming, but they are less essential than in a procedural language; I intended to discuss them at the end of the first semester, but ran out of time.

Data on retention and grades for the past three years CS1/CS2 classes were analyzed. The strongest predictor of whether a student takes CS2, not surprisingly, has always been whether the student was a CS major/minor or just taking the course for interest or distribution requirements. Comparing first-term retention across years, we find small differences:

	Pascal (1997)	Java (1998)	Scheme (1999)
Pass 1st semester	72%	72%	68%
Take 2nd semester	32%	39%	38%

This is perhaps better than expected, since a change in University curriculum in the summer of 1999 encouraged more non-CS-majors to take the course for distribution requirements. Perhaps for this reason, female enrollment is up from previous years. More interestingly, a greater proportion of females passed the first semester than in previous years. The following table shows the makeup of the class as of initial registration, compared with the makeup of the group who passed the course.

	Pascal (1997)	Java (1998)	Scheme (1999)
% female at start	45%	31%	50%
% female passing	36%	28%	54%

(It seems plausible that any gender difference in student performance could be due to instructor personality; however, since the same instructor taught CS1 in Java in 1998 and CS1 in Scheme in 1999, the Java and Scheme columns are comparable.)

An analysis of interaction between student gender and CS1 language in predicting CS1 grade shows significant results. Although in all three years, male students received higher grades (counting A= 6, C= 4, F= 2, drop/withdraw= 1) than female students did, the magnitude of the difference decreased dramatically:

	Pascal (1997)	Java (1998)	Scheme (1999)
Grade-gender correlation	.177	.042	.009

6 CONCLUSIONS AND FUTURE DIRECTIONS

Although the quantitative results to date suffer from confounding factors --- the independent variables of instructor, course language, and incorporation of methodologies like PSP and Pair Programming are closely tied --- it appears that female students have done significantly better in the Fall 1999 class than in either of the previous two years. Better data should be available in a year or two, as all students this year were given an expectations survey at the beginning of CS1 (including, among other things, how many CS classes they plan to take), and as another professor is currently teaching CS1 in Scheme.

Qualitative results, from the instructor s perspective, are clear: the CS1 class in Java was largely about syntax and platform, while the CS1 class in Scheme was largely about data structures, function composition, and software engineering. It felt like teaching.

REFERENCES

[ACM91] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. ACM Press and IEEE Computer Society Press, 1991.

[Bar99] Louis W. G. Barton. "Teaching computer programming by stealth". In *Journal of Computing in Small Colleges,* volume 14, pages 71-85, May 1999. Proceedings of the Fourth Annual CCSC Northeastern Conference.

[BC99] Kent Beek and Ward Cunningham. "Programming in pairs". Web page `c2.com/cgi/wiki?ProgrammingInPairs`, 1999.

[Bei99] John Beidler. "Introducing the Personal Software Process in the freshman seminar". In *Journal of Computing in Small Colleges,* volume 14, pages 202-208, May 1999. Proceedings of the Fourth Annual CCSC Northeastern Conference.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 1994.

[Blu] BlueJ team. "BlueJ". Web page www.pscit.monash.edu.au/bluej/, 2000.

[Fel] Matthias Felleisen. The TeachScheme! project. Web page www.cs.rice.edu/CS/PLT/Teaching/ , 1999.

[FFFK99] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 1999. Also on the Web at www.cs.rice.edu/CS/PLT/Teaching/Lectures/Released/curriculum/.

[HT99]Tom Hilburn and Massood Towhidnejad. "Integrating software quality across the undergraduate computer science program". In *Journal of Computing in Small Colleges,* volume 14, pages 240-241, May 1999. Proceedings of the Fourth Annual CCSC Northeastern Conference.

[Hum98] Watts S. Humphrey. *Introduction to the Personal Software Process.* SEI series in software engineering. Addison Wesley, 1998.

[Wei98] Gerald Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971,1998.

[Wil97] Laurie A. Williams. "Adjusting the instruction of the Personal Software Process to improve student participation". In *Proceedings of Frontiers in Education*, 1997.

[WK99] Laurie A. Williams and Robert R. Kessler. "All I really need to know about pair programming I learned in kindergarten". Submitted to Communications of the ACM; on the Web at www.cs.utah.edu/~lwilliam/Papers/Kindergarten.PDF, 1999.