Controversy on How to Teach CS 1: A discussion on the SIGCSE-members mailing list

Kim B. Bruce*

July 27, 2004

Abstract

A discussion took place on the SIGCSE mailing list in late March of 2004 that raised important issues on how to teach introductory courses using Java. This article attempts to summarize several of the important points raised during this discussion, among them whether or how objects should be taught early or late in a CS 1 course, or indeed whether object-oriented languages should be postponed until a second course.

There has been a tremendous shift to the use of Java in the CS 1 course at colleges and universities in the last several years. Reflecting this change, the Advanced Placement course in Computer Science, designed to provide a college-level computer science course for high school students, moved to Java in the 2003-2004 academic year. However, the use of Java and other object-oriented languages in introductory CS courses seems to be generating a lot of controversy among college faculty. Strong feelings about whether or how Java should be taught in introductory courses erupted in March, 2004, on the SIGCSE mailing list.

Looking back at introductory textbooks using Java, there seems to have been a gradual change in approach over time. The first wave of Java textbooks primarily followed the structure of C++ texts, presenting procedural programming early, while postponing the detailed discussion of object-oriented concepts until after arrays, for example. There has been an increasing trend in more recent texts, however, to introduce classes and objects very early, postponing or even omitting the introduction of some procedural control structures. Papers at SIGCSE and ITiCSE over the last several years have also reflected the same trend.

However, there seems to be increasing concern among some faculty about whether this is the right direction, or even whether it is possible to teach this way successfully. In one of many responses to the announcement of a new ACM Java Task Force (described in the appendix), Elliot Koffman posted a message about the difficulties of getting faculty to adopt new teaching methods involving patterns, graphics, and objects first. He argued that one major reason for this difficulty is that many of the faculty teaching CS 1 do not themselves have the proper training and background in object-oriented programming. Because faculty most often teach the material they feel most comfortable with, they will likely teach procedural programming early.

He also argued that the weaker students entering CS 1 have a very difficult time dealing with the additional layers of abstraction resulting from the use of objects and design patterns, explaining drop-out rates in introductory courses that are claimed to be as high as 30 or 40% in some cases.

*Partially supported by NSF grant CCR-9424123.

He also noted that one of the reasons that GUI components, even classes as simple as `JOptionPane`, are not being used in courses is that students are accessing and developing programs via remote terminals, and because many faculty test programs automatically, resulting in a preference for text-based I/O.

This post led to a very large number of e-mail messages that covered a lot of ground in exploring how we ought to teach introductory courses. One of the more critical posts on new trends in teaching was by Stuart Reges. Reges picked up on an analogy made by Koffman relating new ideas in teaching introductory CS to "the new math", an experiment in the 1960's that attempted to provide a more abstract view of mathematics in elementary school, rather than the more traditional approach that relied primarily on memorization of number facts and algorithms. While there were a number of good ideas in that approach, it ultimately failed due to a combination of poor training of teachers and parents, and excessive abstraction urged by some of its proponents.

Reges argued that the approach involving the early introduction of objects may be "the new CS", complaining that he has struggled to teach this way for five years, and has had little success. He was surprised and dismayed that discussions at SIGCSE have changed almost entirely from *whether* to teach objects early to *how* to teach objects early.

He set down a challenge for those promoting teaching objects early. Show that:

1. a broad range of teachers can teach objects early well,
2. a broad range of students can master it, and
3. the object early approach solves more problems that it creates.

He claimed that the approach of teaching objects early appeared to be failing on all three counts.

It should be of concern to all of us if a recognized talented teacher like Reges has doubts as to whether this "experiment" can succeed after he has tried it for a number of years. Not surprisingly, his posting generated a large number of responses over the next several days.

The responses ranged from agreement to impassioned pleas for alternative ways of teaching this material. As usual with such e-mail discussions, many threads were proceeding concurrently. I'll try to touch on several of the more important streams in this article, though I will be forced by space constraints to omit many. Those who would like to read the original e-mails themselves can find all of the postings at

`http://listserv.acm.org/archives/sigcse-members.html`

Most of the relevant postings are from March 21 to 26, 2004.

1 How and where to fit in objects

A topic raised several times concerned how one can teach object-oriented programming, especially early in the term, and still fit in more traditional topics. A message from Nick Parlante that was posted a few months after the original discussion summarized the challenges well.

He first noted the joys of teaching Java. These include:

1. Concepts like String, ArrayList, etc., are supported directly in the language or standard libraries,

2. Memory safety is built-in, e.g., garbage-collected references, checked array bounds, etc.

3. Support is provided for modularity and design with classes and methods.

However, he then indicated the impossibility of covering all four of the following categories of CS 1 topics:

1. Tactical (low-level) algorithmic code writing (variables, expressions, iteration, logic, etc.).

2. Method decomposition.

3. Class decomposition.

4. Inheritance decomposition.

[I personally would also add topics like dynamic method invocation and subtyping in support of polymorphism.]

Much of the discussion in the earlier thread had to do with the ordering and degree of emphasis on this sort of material in an introductory course using an object-oriented language like Java. The more "traditional" approach is to teach the first two-thirds of the course in the same way one might teach a procedural language (or as some faculty teach C++ as a "better" C), only introducing object-oriented concepts in the last few weeks of the course. Let's call this the *objects-late* approach.

An alternative is to introduce the use of classes and objects very early in the course, with emphasis from the beginning on object-oriented concepts, especially encapsulation of state and behavior in objects, message sending as computation, and dynamic method invocation as a way of choosing between alternatives. We will call this the *objects-early* approach.[1]

Advocates of the objects-late approach argued that loops and conditional statements must logically be taught very early, and that these crucial concepts get less attention than is deserved in the objects-early approach. Jeff McConnell, for example, cited a study[MB02] that he had undertaken with a colleague that showed two trends in CS texts: the size had increased significantly in the last 40 years, and the percentage of space dedicated to the selection and repetition statements, as well as variables and arrays, had decreased.

In contrast, several proponents of the objects-early approach claimed that traditional topics like conditional statements and while loops should be de-emphasized. While the concepts of choice and repetition are clearly important, they can be approached in different ways with object-oriented languages. For example, the selection of alternatives may be accomplished by dynamic dispatch when objects of different classes implement the same interface and respond to the same message (though conditional statements will need to be used with values of primitive types). Some also argued that structural recursion plays as important a role as loops, especially in writing data-driven programs.

A counterargument by some in the objects-late camp was that the if, while, for, etc., must be used when dealing with standard libraries like AWT and Swing, so must be given proper emphasis for students to be successful in Java.

[1] While we will speak of these as though there are two unified camps, there is much variation in views within these camps. We also note that there were several advocates of a functional language-first approach or using a blend of functional and object-oriented languages. We omit discussions of these to keep this article more focussed.

2 The object-oriented paradigm requires a new way of thinking – and teaching

A number of participants brought up the issue of how different the object-oriented approach is from a procedural approach with respect to design and programming. There is anecdotal evidence that it takes quite a while for a procedural programmer to "get" the key ideas of object-oriented programming. Here the issue is not how to use the language constructs, but instead how to *think* about the design process and organization of code. Bjarne Stroustrup, designer of C++, was quoted as saying it takes a proficient C programmer a year to 18 months to become proficient in C++.

A large number of those teaching object-oriented programming in the introductory course are proficient with object-oriented concepts, but others are simply thrown into an introductory Java course, even though their main experience is with procedural programming. Quite naturally they will tend to teach most of the course the way they always have, including object-oriented topics where they occur in the text or syllabus, but without rethinking the overall approach of the course.

Several participants in the discussion emphasized the importance of faculty having enough experience in object-oriented programming to effectively understand the differences in mind set with procedural programming. For example, short object-oriented programs do not provide that mind set. Faculty need to have developed programs larger than those assigned in introductory courses in order to really understand why the organization supported by the object-oriented style is valuable. Once that understanding is there, the style can be taught more effectively to novices, even on relatively small programs.

In the same posting mentioned earlier, Reges complained:

> I think that the reason we're having this heated debate is that it hasn't turned out that a broad range of teachers have been able to easily teach this effectively. We hear things like: Professor A has succeeded because he uses a custom IDE designed specifically for teaching objects; Professor B has succeeded by developing a framework of graphics classes; Professor C has succeeded by developing tons of supporting code for each assignment. These are bad signs, not good signs. Where is the list of professors who have succeeded because the material is straightforward to teach? And if the material isn't straightforward for a lifelong computer scientist to teach, then can it really be all that fundamental?

Proponents of objects-early like Joe Bergin countered that "if you want to teach a new paradigm ... then you need to use new pedagogy. ... [Y]ou can't teach o-o with a piece of chalk. You need to scaffold the process just as your three professors (A, B, and C) have done."

In a later post, Richard Rasala stated:

> Let me summarize what I think are the "abstract" wishes of many in the pedagogical community vis-a-vis teaching o-o.
>
> 1. We want to teach object-oriented thinking with the simplicity and elegance we remember from Wirth's classic [Pascal] text.
> 2. We want simple examples that take only a few pages of code but which nevertheless illustrate the power of object-oriented ideas.
> 3. We want classes with rich behavior that are easy to use.

4. We want collections of classes that interact nicely and may be the basis for case studies.
5. We want to use a widely available o-o language rather than a research language used only in universities.

This is what leads us to the choice of Java.

6. Now, having chosen Java, we want to use it "as is" without any pedagogical support other than that supplied by the vendor via the vendor's commercial libraries.

After noting that we can't have all 6 of these, he goes on to say:

> "...[T]here appear to be only three forms of support that have been discussed in the SIGCSE literature:
>
> 1. Pedagogical IDEs
> 2. Microworlds
> 3. Encapsulated Java tools for pedagogy."

Pedagogical IDE's for Java include BlueJ and DrJava. They provide simplified environments designed especially for novices.

The best known microworld is "Karel J. Robot" [BSRP04], but "Alice" [CDP03], "Buggles and Bagels" [TRSH99], and the Marine Biology Simulation Case Study designed by Alyce Brady for the Advanced Placement Program in Computer Science [Boa03] are other examples of microworlds.[2] By programming within the context of a microworld, a student can work with a rich set of primitive objects that can be used to illustrate the key concepts of object-oriented languages.

There are several collections of Java tools designed to make programming easier for novices. These provide simplified features (e.g., GUI components) or provide rich collections of classes (e.g., graphics) that are easily usable by novices. They include the Java Power Tools [RRP01], a Widget [RP98] graphics library, and our own objectdraw [BDM01] library.

Alyce Brady cited psychological studies that claimed that most people learn more easily by seeing concrete examples first and then generalizing. The microworlds and graphics libraries provide such concrete examples, as do visible behaviors invoked as the direct result of events. Interestingly, these studies also showed that most faculty fall into the minority of the population that seems to learn best via abstractions and principles first, followed by concrete examples. That may help remind us that our first intuitions as faculty on how to present material may not always be correct.

While some people appear to have been successful without using one or more of the pedagogical items described above, it appears that many faculty have been having difficulty in figuring out a strategy to use in teaching languages like Java to novices. While many instructors would have difficulty finding time to design, implement, and support a pedagogical IDE, microworld, or set of libraries, many of the tools cited above seem to be quite stable and well-supported. See also the discussion in the appendix of the new ACM Java Task Force for the development of a set of tools that will be supported by ACM.

Finally, Brady raised the question of why so few CS instructors research and take advantage of the work other people are doing in computer science education. We would all be better off if we could leverage the work of other instructors in our own courses.

[2] The COOL program [BFG+03] initiated by Kristen Nygaard may also be seen as being based on a microworld.

Interestingly, the community of Advanced Placement CS teachers seems to be an exception to this tendency to work in isolation. The AP CS course, which is intended to provide one or two semesters of college-level work in Computer Science for high school students, changed languages from C++ to Java in 2003-2004. While many high school teachers took advantage of Java workshops, they also rely heavily on a mailing list run out of the College Board. On this list, the teachers trade tips on books, tools, library packages, how to use IDEs with particular tools, how to cover particular topics, exams and homework, as well as raising and answering questions on peculiarities of Java.

While the SIGCSE mailing list occasionally contains discussions on these issues, it would likely be annoying to many subscribers if there were ten or more messages per day addressing issues in teaching introductory courses. Perhaps it would be helpful to set up a special mailing list for those interested in discussions of teaching introductory courses using an object-oriented language.

3 My views

The questions raised in this discussion, including those posted but not included in this summary, indicate challenges that faculty must confront in teaching an object-oriented language like Java. In this section I will present my own views on these issues, paraphrased a bit from my contributions to the on-line discussion.

At Williams we developed the objectdraw library to support an objects-early approach to the teaching of Java in our introductory course. Our library contains classes for geometric objects, including both text and images. It also provides syntactic support for event-driven programming using mouse actions. The style of programming is the same as the standard Java event model, but we eliminate much of the syntactic overhead for beginners. The combination of graphics and simple event-driven programming allows us to write interesting programs using graphics objects from the first week of classes. Thus students are using the object-oriented style from the very beginning. Later in the course students learn the standard Java event-handling style when we discuss the use of GUI components.

Our course also includes the early introduction of simple concurrent processes via an `ActiveObject` class that differs from `Thread` primarily by providing a pause method that does not throw exceptions. Experts know that spawning separate processes is necessary to keep an event-driven system responsive, but it also provides us with an effective way of performing simple animations. We also cover structural recursion before discussing arrays.

What do we omit in order to do these extra things? Not much, as it turns out. The graphics and event-driven programming provide enough pedagogical advantages that the time to teach them is more than compensated for in the ability to present ideas about message sending and parameters in simple consistent examples from the beginning. Because we handle concurrency only where the interactions between processes are relatively simple, we end up mainly using them for examples involving animations of graphics in which students gain a lot of practice using while loops. We tend to cover specific Java constructs in a "just-in-time" fashion. Thus we cover while loops just before covering animations, interfaces before introducing GUI components and the use of listeners, and for loops just before covering arrays. We probably spend a bit less time than we used to on different programming strategies using loops, but our students don't seem to miss it.

We rely heavily on the use of interfaces for driving home dynamic method invocation. On the other hand we usually cover inheritance fairly lightly, and typically late in the course. Files are also

covered fairly lightly (if I had my druthers, I'd cut them even further). We don't teach much class design, instead trying to illustrate good design, with the understanding that students will work on design in the follow-up courses.

This approach has worked well for us and those who have adopted our materials. However, there are many other options. Roughly, there are three basic approaches that faculty teaching object-oriented languages (and Java in particular) in introductory courses can try:

1. Use an objects-late approach, but tinker at the edges for minor improvements.
2. Try one of several objects-early approaches using pedagogical tools. These include pedagogical IDE's like BlueJ and DrJava, libraries like Java Power Tools or objectdraw, or microworlds like Karel J. Robot. The resulting approaches are quite different from each other. Talk with others who have tried one of these and see what works for you.
3. Teach a non-object-oriented language for the first course. This could be procedural, e.g., C, C++, Pascal, Modula-2, or Ada; or it could be functional, e.g., Scheme, LISP, ML, or Haskell. By graduation, students should be comfortable with at least two different kinds of languages anyway. However, be aware that many faculty believe that it is harder to transition to good object-oriented style if the first exposure is to a procedural language.

When my colleagues and I started looking at teaching Java in our introductory course (having several years of experience using Java in our data structures course), we didn't see how we could teach an object-oriented language using traditional approaches, especially because of the need for good examples of usable objects from the beginning. We started out looking at the microworlds approaches, but eventually went off in our own direction designing object-oriented graphics classes. It took a lot of planning and lots of work on building supporting libraries, but it worked exceptionally well for us and for others who have used our approach. Today there are several groups that have built interesting libraries or microworlds, so the overhead for those who want to try an objects-early approach with supporting tools is not as great. However, teachers will have to figure out how to make the paradigm shift in their own teaching in order to adopt one of these approaches, and that is not always easy to do.

Clearly my favorite alternative from above is (2), and I think several of the non-traditional approaches have a great deal of promise. My next choice would be (3), with (1) a distant last. I think you do a disservice to students by trying to teach them in a style that is not a good fit with the language being used.[3] While tinkering may help, it is likely to result only in small improvements. There is a price to be paid for starting with a different paradigm and then making the shift later to an object-oriented language, but at least students will get a solid (and successful!) introduction that will teach them important ideas that they will use later.

4 What next?

Where do we go from here? There are anecdotal reports of departments that have given up teaching Java in introductory courses, generally returning to using C or C++, though some may choose instead to to teach the first course in a different object-oriented or functional language.

[3]One contributor to the discussion pointed out an Australian survey that showed a mismatch between the languages taught and the programming paradigm taught. Over 80% of classes used an object-oriented language, but more than half of the faculty chose to teach using the procedural paradigm.

The one thing that there was near universal agreement on during the discussion is that it is a challenge to teach objects early. Many of those who have successfully used the objects-early approach to teaching introductory courses using pedagogical tools or libraries strongly encourage other faculty to experiment with these tools, whether pedagogical IDEs, special libraries providing useful classes, or microworlds. Unfortunately, there is currently insufficient data to evaluate how effective these tools are in teaching introductory courses. With more systematic experimentation we can hopefully provide an answer to Reges' three-part challenge to faculty and discover more effective ways of educating novice programmers.

A particular problem has been the lack of texts that are based on using these tools.[4] However, there are now texts that make explicit use of BlueJ [BK04], and more are on the way that use this and other IDEs and pedagogical libraries.

As some remarked during the on-line discussion, educational experiments almost always succeed because of the enthusiasm of the experimenters. We need more people to attempt to duplicate the success of those claiming success with their approaches. It will not be easy because teaching in these new ways often requires a very different approach and even a different way of thinking about the material.

Revolutions, even minor ones involving ways of teaching introductory courses, result in a great upheaval (i.e., lots of work), and certainly are not always successful. It is quite understandable that many faculty are not willing to make major changes in the way they teach their introductory courses without some guarantee that they and their students are likely to be successful. Indeed it may be that the majority of faculty revert to previous ways of teaching using procedural or functional languages. However, it would be a shame to go back if very successful approaches are available.

It would help if a group of experts in educational research were to design experiments that will allow faculty to examine the success of the innovative approaches proposed for teaching Java in CS 1. It would also be useful to know if there are features specific to Java that make it difficult to teach, but might not be present in other object-oriented languages. If that is indeed the case, then it would be tempting to propose a pedagogically-oriented object-oriented language. Sadly, in spite of the success of Pascal, a great majority of faculty today would likely resist using such a language even in an introductory course.

In the meantime, what can we do as faculty? Certainly one can learn about these new and different approaches by reading articles in CS education journals and conference proceedings. Perhaps a special session at an upcoming SIGCSE meeting could be designated for faculty espousing these approaches to present their materials and their rationales.

The AP CS mailing list was mentioned earlier as a very useful resource for high school AP teachers. Small private lists on teaching CS 1 with objects have existed in the past, e.g., CS1OBJ-L and its successor, OOTINCSE. However these mailing lists no longer are very active and are certainly not well known in the community. While the main SIGCSE list is useful for general conversations, perhaps SIGCSE could facilitate setting up a specialized list for those wishing to discuss how to teach CS 1 using Java or other object-oriented languages.

Finally, the ACM Task Force on Java is already at work on trying to provide support for instructors using Java in introductory courses. The appendix provides a brief description of this

[4] In the first wave of Java texts, texts that were dependent even on the use of a few classes to simplify I/O had difficulty in getting acceptance in the marketplace. Many of these authors turned to a pure Java approach, running into the difficulties of using an industrial-strength language designed for professionals.

task force and the ways in which it intends to provide support for instructors using Java in the introductory course.

We are in one of those periodic difficult times where we are struggling to find an effective way of teaching introductory courses that will provide a strong education for our students. Ideally the experimentation would have been carried out well before lots of schools began a shift to Java. Clearly, however, dissatisfaction with the old languages and techniques for teaching, as well as the excitement of Java, carried people to make changes before many were ready. While this has resulted in problems, I have every confidence that we will indeed yet again find effective and interesting ways of teaching the introductory course in computer science.

Acknowledgements: Thanks to all of those who participated in the on-line discussions on teaching. My apologies to the majority of contributors whose views could not be summarized in this article. Thanks to Andrea Danyluk and Eric Roberts for comments on an early draft of this paper.

References

[BDM01] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 6–10. ACM Press, 2001.

[BFG+03] O. Berge, A. Fjuk, A. K. Groven, H. Hegna, and J. Kaasbll. Comprehensive object-oriented learning - an introduction. *Journal of Computer Science Education*, 13(4):331–335, 2003.

[BK04] David J. Barnes and Michael Klling. *Objects First with Java: A Practical Introduction using BlueJ.* Prentice Hall /Pearson Education, 2nd edition, 2004.

[Boa03] The College Board. Marine biology simulation case study. available on-line, 2003. http://www.collegeboard.com/student/testing/ap/compsci_a/case.html.

[BSRP04] Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis. *Karel J. Robot: A gentle introduction to the art of object-oriented programming in Java.* Unpublished, 2004.

[CDP03] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195. ACM Press, 2003.

[For04] ACM Java Task Force. Taxonomy of problems in teaching Java (draft), 2004. http://www-cs-faculty.stanford.edu/∼eroberts/java/java-problem-taxonomy.html.

[MB02] Jeffrey J. McConnell and Debra T. Burhans. The evolution of CS1 textbooks. In *Proceedings of the Frontiers in Education*, pages T4G–1–T4G–6, 2002.

[Rob04a] Eric Roberts. The dream of a common language: the search for simplicity and stability in computer science education. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education*, pages 115–119. ACM Press, 2004.

[Rob04b] Eric Roberts. Resources to support the use of Java in introductory computer science. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education*, pages 233–234. ACM Press, 2004.

[RP98] Eric Roberts and Antoine Picard. Designing a Java graphics library for CS 1. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 213–218. ACM Press, 1998.

[RRP01] Richard Rasala, Jeff Raab, and Viera K. Proulx. Java power tools: model software for teaching object-oriented design. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 297–301. ACM Press, 2001.

[TRSH99] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before iteration in CS1. *The Journal of Computing in Small Colleges*, 14(4):86–101, 1999.

A ACM Task Force on Java

Aside from the general issues discussed in the earlier portions of this paper, there have also been other discussions in SIGCSE-sponsored forums of specific issues related to teaching introductory courses in Java. A paper by Eric Roberts[Rob04a] in SIGCSE 2004 discusses challenges to using Java as a teaching language, including issues of complexity and instability. Many of the problems identified are shared by many industrial strength languages because they are specifically defined to be used by professional programmers, rather than as teaching tools for novices.

In partial response to the problems identified with Java, Roberts has championed the creation, and is now chair, of a task force[Rob04b] to identify and develop resources to make it easier to teach Java in introductory courses.[5] The Java task force has provided a draft document [For04] that enumerates most of the difficulties in using Java for introductory courses that have been cited over the years. It also includes a bibliography of papers discussing these issues. The task force has identified a subset of these problems that it will attempt to remediate by developing or identifying (and supporting) an appropriate collection of Java-based teaching resources, and by identifying a collection of classes from the standard Java libraries that are likely to be most useful in an introductory course. An important part of the job of the task force is to create a mechanism that will guarantee support of the teaching resources created as a result of the task force's actions. That way faculty can be assured that courses developed using these materials will be supported in the future.

The task force has an ambitious deadline of completing this task by June of 2005. Contributions have already been solicited by the task force, and a draft report will be presented for discussion at SIGCSE 2005.

[5] Disclosure: The author of this paper is a member of the task force.