The Dream of a Common Language: The Search for Simplicity and Stability in Computer Science Education

Eric Roberts
Stanford University
eroberts@cs.stanford.edu

ABSTRACT

In recent years, the languages, paradigms, and tools used to teach computer science have become increasingly complex. This added complexity puts pressure on designers of introductory courses, who must cover more material in an already overcrowded syllabus. The problem of complexity is exacerbated by the fact that languages and tools change quickly, which leads to profound instability in the manner in which computer science is taught. The situation has reached a point where it is difficult for individual computer science teachers to keep up. This paper examines the factors that promote complexity and instability in computer science. It then goes on to argue that we, as educators, must take responsibility for breaking this cycle of rapid obsolescence by developing a stable and effective collection of Java-based teaching resources that will meet the needs of the computer science education community. Such an initiative is already in progress under the direction of a special task force appointed by the ACM Education Board. The charter and current plans for that task force are being presented in a special session at this symposium.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: computer science education, curriculum.

General Terms

None.

Keywords

Computer science education, CS1, teaching libraries, Java.

1. INTRODUCTION

In recent years, introductory computer science courses have become much more advanced in terms of both the level of conceptual material presented and the sophistication of the programming tools used. To a large extent, this change reflects advances in the discipline of computer science and therefore represents a positive educational development. Students today would be bored to tears if they had to learn computer science the way it was taught a generation ago. Although learning how to average a list of numbers might once have seemed exciting to those of us who are now senior faculty, it holds little attraction for students who have grown up in the age of Nintendo, the Palm Pilot, and the Internet. The ubiquity of computing in daily life creates new expectations for computer science courses that lead us to cover topics like graphical user interfaces and distributed computing, even at the introductory level. Including these concepts in the first-year syllabus in turn forces institutions to adopt new programming languages and paradigms that make it easier to teach that material.

While the growing sophistication of first-year computer science courses has made for a more exciting curriculum, the change has not come without a certain amount of pain. Introducing new, more advanced concepts requires more time, which makes it more difficult to cover the necessary material in the traditional two-semester sequence. The *Computing Curricula 2001* report [12] describes the problem as follows:

> The number and complexity of topics that entering students must understand have increased substantially, just as the problems we ask them to solve and the tools they must use have become more sophisticated. An increasing number of institutions are finding that a two-course sequence is no longer sufficient to cover the fundamental concepts of programming.

The CC2001 report goes on to recommend the adoption of a three-course introductory sequence to "cover the growing body of knowledge in a way that gives students adequate time to assimilate the material."

Unfortunately, extending the introductory sequence to three courses is unlikely to solve the full range of problems associated with increasing sophistication at that level of the curriculum. Along with an increase in the sophistication of the concepts—which is an intrinsic and ultimately positive characteristic of a dynamic discipline—computer science faces two additional challenges in areas in which dramatic increases have had a more negative impact on pedagogy:

1. *Complexity.* The number of programming details that students must master has grown much faster than the corresponding number of high-level concepts.

2. *Instability.* The languages, libraries, and tools on which introductory computer science education depends are changing more rapidly than they have in the past.

These simultaneous increases in complexity and instability have made it difficult for computer science education to keep up with the pace of change. On the one hand, the additional complexity that now adheres to the introductory curriculum means that it takes longer to develop effective pedagogical materials, strategies, and tools. On the other, the increasing pace of change means that such materials more quickly become obsolete. Together, these forces threaten our collective ability to offer effective computer science education.

2. ACCIDENTAL AND ESSENTIAL COMPLEXITY

To a certain extent, increasing complexity and rapid change are natural concomitants of any vibrant new technology. As a result, some may argue that these characteristics are an inevitable consequence of our desire to use modern, object-oriented techniques. This conclusion, however, is by no means obvious. Although, as described later in this paper, there are economic and technical forces that encourage complexity and instability, there is

no *a priori* reason to conclude that it is impossible to teach the essential concepts of object-oriented programming in an environment that is both simple and stable—at least in comparison to the languages and tools now in place.

To illustrate this point, it is useful to reintroduce the distinction between essential and accidental characteristics that Fred Brooks adopts in his landmark paper "No silver bullet: Essence and accidents of software engineering" [3]. Following Aristotle, Brooks defines as *essential* those characteristics that are inherent in the nature of some entity, in contrast to *accidental* characteristics, which are those that attend the current realization of that entity but are not intrinsic to it. Within this framework, examples of essential complexity include encapsulation, inheritance, reusability, interface design, and the various other conceptual aspects of object-oriented programming. The complexity associated with these concepts is therefore a necessary consequence of our collective decision to teach in a modern, object-oriented style. By contrast, an example of accidental complexity is the fact that there are on the order of 50,000 public methods in the Java 2 class libraries. Similarly, the continuing rapid obsolescence of particular languages, libraries, and tools is also an accidental property of modern computing, related more to the economics of the industry than to any essential aspect of computer science itself.

My fundamental thesis in this paper is that the computer science education community must find a way to separate the essential characteristics of object-oriented programming from those accidental features that are merely artifacts of the particular ways in which technology is implemented today. If we succeed in doing so, we can eliminate much of the inessential complexity and instability that plague computer science education today, which will allow more time to cover the essential elements of the subject matter.

3. ILLUSTRATING THE PROBLEM: AP COMPUTER SCIENCE

As an example of the pitfalls caused by the combined forces of complexity and instability, consider the history of the Advanced Placement (AP) program in computer science promoted by the College Board, a nonprofit educational organization in New York City. The AP Computer Science (APCS) exam was introduced in 1984 when the language of instruction—both for APCS and for the vast majority of colleges and universities in the United States—was Pascal. That situation remained stable for over a decade, after which changes in both the practice of the discipline and the introductory curricula of many computer science programs made it clear that Pascal was no longer broadly suitable as the basis for APCS. In 1995, the College Board announced that the APCS program would shift to C++ in the 1998-99 academic year, and the first C++ exams were administered in May 1999.

The C++ regime, however, lasted for a much shorter time than the Pascal period that preceded it. In the four years it took between making the decision and completing the implementation of the language change, the evolution of computing technology continued unabated. Java, a new object-oriented programming language that had only just been introduced when the AP announced its C++ decision quickly gained a foothold in the educational community and then began to blossom over the next few years. By the time the C++ exam was put in place, conventional wisdom held that Java, and not C++, was in the ascendant as the primary language for introductory computer science courses. In 2001, just two years after implementing the change to C++, the College Board announced that the APCS program would move to Java beginning in 2004.

This change, following as it does so closely on the heels of the previous transition, has had a disastrous impact on secondary schools that teach the APCS program. Teachers who are not yet comfortable with C++ suddenly find themselves faced with the prospect of learning yet another language in a short amount of time. As a result, many secondary schools—and even some entire states—have dropped the APCS program from their curriculum. Even in programs that continue, teachers feel a high level of anxiety about the new material. A recent ACM survey, for example, found that

> At present 79% of high school computer science teachers rated their current knowledge of Java as poor to fair and only 3% rating their knowledge as excellent. At the same time, 86% rated their personal need to learn Java as very important to critical and 89% indicated that they needed to do so within one year. [14]

Unfortunately, the resources available to support retraining of high school computer science teachers are insufficient to meet the demand. Teachers who need significant training in the theory and methods of object-oriented programming are being forced to settle for one-and two-day workshops that emphasize Java syntax or highlight the specific differences between Java and C++. Through no fault of their own, these teachers will soon return to classrooms dramatically underprepared to teach high-level concepts of object-oriented programming.

Even for those school systems that succeed in making the transition, other worries remain. How long will Java maintain its relative dominance as the preferred language of instruction? Are there other languages, such as C#, waiting in the wings to take its place? Are school systems in danger of having to go through yet another massive transition two or three years down the line?

The problem of providing teachers with adequate training against a backdrop of instability was recognized even at the time of the shift from Pascal to C++. In July 1995, six prominent computer science educators sent a letter to *Communications of ACM* arguing against the proposed language change. Their argument is based in part on the complexity and instability of the proposed new paradigm:

> Pascal, the current AP language, is compact and designed around a small, coherent set of principles. C++, in contrast, was designed to meet industrial rather than pedagogical goals. It does not have a simple, coherent, conceptual model. It is difficult for teachers to present it consistently through a course without talking a lot about exceptions. Moreover, switching to C++ isn't just switching languages—it is switching to a programming paradigm of object-oriented programming and object-oriented design. That is a huge change. Even universities and colleges are still grappling with it, and the material is both unstable and tricky to teach. . . . At the very least, teachers and students will be forced to work with an enormously complicated tool, large portions of which will be mysterious to them. These mysteries of the language will inevitably rear their heads as students write and debug programs, and there is little experience with or understanding of how much of C++ a teacher needs to master in order to use it as a teaching tool. [1]

Even though Java is simpler and conceptually cleaner than C++, many of these same arguments apply to teaching Java in secondary schools. If one includes the standard library classes along with the language itself, Java is wildly more complex than Pascal was. Kathleen Jensen and Niklaus Wirth's classic *Pascal User Manual and Report* [7], which included both a tutorial and a language definition, runs to 266 pages including all appendices and the index. By contrast, the popular *Java How to Program* book by Harvey and Paul Deitel [4] is a massive tome of 1536 pages—more than a bit daunting even to students raised on Harry Potter. These books, moreover, have a very short revision cycle because of the rapid changes in the language. While Jensen and Wirth's book went through two new editions in 12 years, leading

Java books must be updated every one or two years. Cay Horstmann and Gary Cornell's *Core Java* [6] is in its sixth edition just seven years after its initial release.

As perhaps an even more dramatic illustration of the growth in complexity, it is sobering to realize that there are more public methods in the **`java`** and **`javax`** package hierarchies than there are *words* in Jensen and Wirth. The amount of text once deemed sufficient to teach the standard introductory programming language is thus no longer sufficient for a full index of the operations available today.

Of course, no one at the introductory level—or even at the advanced level for that matter—needs to learn about all the classes and methods available in the official collection of application programmer interfaces (APIs) that Java defines. These APIs represent a huge collection of tools from which teachers and students can select the subset they need, just as professional programmers do. The problem, however, is that the complexity of those APIs is immediately visible because they are part of the public framework available to Java programmers. Under pressure from publishers and the professional marketplace, authors of Java books tend to err on the side of including too large a subset so as to fend off criticism that their treatment is any way incomplete. More importantly, the HTML pages for these APIs produced by **`javadoc`**, which are designed to be the primary documentation for the standard APIs, cover the entire collection of available classes and methods without giving any indication of which ones are essential. The visibility of this enormous mass of detail has a negative impact on both students and teachers. For students, and particularly those who are insecure about their level of expertise or whose learning styles make it difficult for them to proceed on the basis of partial knowledge, this level of detail is overwhelming. For teachers, particularly those whose formal training in computer science is weak or nonexistent, the problem may be even greater, since the documented existence of a vast collection of mysterious routines and tools serves to reinforce a lack of confidence. Students will undoubtedly try to use the obscure methods they find in the **`javadoc`** pages and ask their teachers any questions that arise. Knowing that they are completely unprepared to answer questions about perhaps 95 percent of the API methods must produce anxiety in teachers, who are likely to adopt the defensive posture of focusing on syntax, imperative semantics, and other concepts well within their domains of expertise at the expense of more important conceptual material.

4. FACTORS ENCOURAGING COMPLEXITY AND INSTABILITY

Finding a way to reduce the level of detail complexity and mitigating the effects of rapid change would be extremely beneficial to computer science education. For any such attempt to succeed, however, it is important to understand why computer science is so heavily affected by these issues and why things seem to be getting progressively more complex and unstable, as opposed to settling into a calmer, more mature phase. The subsections that follow explore some of the factors that seem important in this regard.

4.1 The vitality of the discipline

The most obvious factor encouraging the complexity and instability of computer science is the vitality of the discipline. Since the time of Pascal's academic heyday 25 years ago, the size of the community working in software development has increased substantially. As a result, there are more people and commercial interests with a stake in the progress of computing technology and the expertise necessary to influence its direction. When a promising new technology such as Java emerges, those individuals and companies invest time and energy in promoting the development of that technology. That investment creates pressure for the incorporation of new features and frequent new releases of the language and its supporting APIs.

From this perspective, the rapid pace of change in Java and its APIs can be seen largely as a sign of its success. If Java has evolved more quickly than its predecessors, the reason is certainly in part the result of the fact that so many people became excited about the language and pushed its evolution. The attractive aspect of this piece of the rationale is that the instability in Java is likely to moderate over time. As the flurry of excitement over the novelty of the language subsides and the language continues to mature, the hope is that the frenetic pace of change will begin to slow, creating greater stability for its users. As of yet, however, there are few if any signs of such a slowdown.

4.2 The economics of proprietary languages

Although the excitement surrounding Java has helped to promote its rapid development, there are other, more subtle reasons why the languages that are popular today evolve differently than their predecessors. In the past, most programming languages used for teaching—including FORTRAN, BASIC, Pascal, Lisp, Scheme, C, C++, and Ada—were not owned by a particular company. Although specific compilers and tools were proprietary, the languages themselves remained part of the public domain. The situation is different with Java, which is owned by Sun Microsystems. On a day-to-day basis, the fact that Java is a proprietary language has little practical significance for individual users. Over the longer term, however, the fact that Java development is driven in large measure by corporate economic considerations has a significant impact on its evolution that ends up affecting its user community.

Like any software company, Sun Microsystems seeks to promote the use of its products within an ever-expanding market. The more programmers use Java, the greater the incentive for other companies to license Java technology from Sun. Any company whose products incorporate the Java Virtual Machine or other aspects of Sun's proprietary software base pay Sun for the privilege, and it is in Sun's interest to maximize that user base. The economics of software development, however, place relatively little value on stability as opposed to the constant introduction of new features. Stable software does not generate a constantly renewing revenue stream. To sell new versions of a product to an existing customer base, companies must continually offer new functionality to make the new versions attractive. And even though the economics of a base technology like the Java Virtual Machine are different from those that apply to more user-centric products, the corporate culture of the industry promotes continuous redevelopment. The frequent release of new versions of Java, each of which incorporates a huge number of new features, is in some sense the most direct cause of Java's complexity and instability.

The increasing centrality of proprietary languages also has the effect of making the user community more subject to the vicissitudes of the market. Consider the recent appearance on the educational scene of C#, a language that has been proposed by members of the SIGCSE community as an appropriate successor to Java [11]. C# is directly associated with Microsoft, which—independent of any strengths and weaknesses in the language—automatically makes it a contender one can ill afford to ignore. Microsoft is the undisputed leader of the software industry and has a historical propensity for using its dominant position in the operating system market and its massive supply of ready cash to promote its other products. By contrast, the fortunes of Sun have sagged to the point that industry analysts have begun to speculate on its potential demise [5]. Microsoft has a strong

economic incentive to eliminate Sun as a competitor and would hardly mind killing off Java in the process. Given Microsoft's success in other corporate battles and the apparent unwillingness of the U.S. courts to impose effective antitrust sanctions, there is little reason to doubt that it will accomplish that task.

In fact, even if Sun were to go into eclipse, Java would probably survive. Other players in the computing industry, most notably IBM, have a strong investment in Java and would probably take over the maintenance of Java if Sun were to fail. Even so, such a change would almost certainly increase the instability of Java as its new owners sought to adapt it to their own business plans. The crux of my argument is simply that the economics of the industry tend to promote instability in proprietary languages, no matter whether that instability comes from the conventional incentive of a company to market new versions of its products or from the more drastic effects of corporate failure and reacquisition.

4.3 Unintended consequences of encapsulation

The economic forces promoting complexity and instability outlined in the preceding section provide the major justification for the conclusions I draw later in this paper. In the interest of completeness, however, it is useful to consider the possibility that some of the complexity and instability might also have a technical source.

From my experience programming in Java over the last several years, I've become increasingly convinced that the rigid enforcement of encapsulation the language provides—useful as it is in terms of protecting disparate parts of a program from interfering with one another—does have the unintended consequence of promoting both complexity and instability in package design. The essence of the problem is that rigid encapsulation often makes it difficult for clients of a particular class to extend its functionality.

At first, this assertion seems inconsistent with the idea that object-oriented languages derive their greatest strength from their extensibility. If a client wants to enhance the functionality of a class, object-oriented languages like Java make it easy to create subclasses that have all the power of the original but which are enhanced by new methods that extend the base functionality. This idea is wonderful in theory, but often fails in practice, particularly when the creation of the objects that you wish to use is not under your direct control. It is difficult, for example, to extend the capabilities of a class like **`Graphics`**, because **`Graphics`** objects are instantiated by component classes that provide no easy access to the mechanism by which those instances are created. While it is sometimes possible to use factory methods in Java library classes to achieve the appropriate effect, the standard APIs are inconsistent in their design and do not always provide the necessary hooks.

A similar situation occurs when one encounters—as one all too often does—a bug in the implementation of a Java API. Because the source code is typically part of the Java distribution, it is usually easy to locate the problem. Unfortunately, it is usually not possible to fix that problem by any straightforward strategy available to the client. As often as not, the access rules provided to ensure encapsulation prohibit the client from overriding the offending method in order to circumvent the bug.

So what are the options for the client who identifies a critical piece of missing functionality or a showstopping bug in one of Sun's library classes? Since it is typically impossible for the client to fix the problem independently, there are two remaining options:

1. Encourage Sun to fix the bug or incorporate the missing functionality in the next release. While this approach is not always successful, it does dovetail well with Sun's natural economic incentive to issue periodic new releases.

2. Abandon the existing class and develop an entirely new mechanism that has the desired behavior. Although this course of action might seem difficult, it occurs quite frequently in the industry. The Swing libraries, for example, got their start as an independent effort to fix the inadequacies of the original **`java.awt`** package and were only later incorporated into the official release.

Both of these strategies feed Java's instability and lead to the proliferation of new classes and methods.

The fact that rigid encapsulation can make it difficult for individual clients of a package to add new functionality also has an effect on the strategy used to design APIs. If extension is relatively straightforward—even if the mechanism for extension is simply to introduce new independent procedures in a language like C—the designer of an API can opt for the simple approach of providing a minimally sufficient set of public methods under the assumption that any missing methods can be added by extension. By contrast, if extending the original functionality is difficult, designers have an incentive to include every method that any client might conceivably want, even though such an approach adds to the complexity of the design.

5. WHERE DO WE GO FROM HERE?

Beset by the twin forces of complexity and instability, computer science education today faces an increasingly difficult challenge that threatens to become a crisis. The languages and tools we use today in introductory courses change much more rapidly than their predecessors in the early years of computer science. As a result, computer science textbooks and curricula have a shorter shelf-life and must be replaced every year or two. This instability, coupled with the increased detail complexity of the languages and tools, often makes it difficult to develop new materials before they become obsolete.

Fortunately, neither the complexity nor the instability currently associated with languages like Java is essential to our discipline. There is every reason to believe that one can offer an excellent introduction to computer science and object-oriented programming in a much simpler and more stable environment. Indeed, many existing Java textbooks [8, 9, 13], available tool packages [10], and the AP Computer Science program [2] define Java subsets and simplified tool packages that offer a less complex, and presumably more stable, platform for teaching. Such subsets and packages, however, often face rejection from potential adopters on the grounds that they are not "standard." Our collective hope has been that Sun—as the guardian of the Java language—would develop a standard introductory package explicitly designed for pedagogical use. In each of the last three years, there has been a flurry of messages on the **`SIGCSE.MEMBERS`** list bemoaning the fact that Sun appears to have little interest in developing such a package.

I believe it is time to revisit the question of how we define the notion of "standard." The principal advantage of a standard is that the authority behind the standard offers some guarantee of correctness and stability. As evidenced by the various ANSI standardization efforts, that guarantee can be effective when the body setting the standards represents the interests of the user community. The force of that guarantee, however, is compromised when the standard is subject to the economic interests of a particular vendor. The fact that there has been a succession of Sun-endorsed Java standards has not provided any semblance of stability or any relief from the growing complexity that makes introductory programming so difficult to teach.

What we need instead is a community-sponsored effort to develop our own standards. In much the same way that the ACM Education Board has assembled task forces to develop curricular recommendations such as CC2001, it should be possible to

convene a similar task force to define a simple, stable subset of Java and a set of supporting libraries that can meet the needs of our community. If such an effort involves broad representation from the affected constituencies and can attract sufficient funding, it will be of enormous valuable to the computer science education community.

In the fall of 2003, the ACM Education Board did indeed convene a task force along these lines with the following charter:

> To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

The structure and current plans for that task force are the subject of a special session at this symposium.

REFERENCES

1. Hal Abelson, Kim Bruce, Andy van Dam, Brian Harvey, Allen Tucker, and Peter Wegner. Letter opposing APCS change. *Communications of ACM*, June 1995.
2. Owen Astrachan, Robert (Corky) Cartwright, Gail Chapman, David Gries, Cay Horstmann, Richard Kick, Frances Trees, Henry Walker, and Ursula Wolz. Recommendations of the AP Computer Science ad hoc committee, October 2000.
3. Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, April 1987.
4. Harvey M. Deitel and Paul J. Deitel. *Java How to Program* (fifth edition). Englewood Cliffs, NJ: Prentice Hall, 2002.
5. *The Economist.* Setting Sun? Unattributed article in the business news summary from July 26, 2003.
6. Cay S. Horstmann and Gary Cornell. *Core Java 2—Volume I: Fundamentals* (sixth edition). Englewood Cliffs, NJ: Prentice Hall, 2002.
7. Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report* (third edition). New York: Springer-Verlag, 1985.
8. Elliot B. Koffman and Ursula Wolz. *Problem Solving with Java*. Boston, MA: Addison-Wesley, 1998.
9. John Lewis and William Loftus. *Java Software Solutions: Foundations of Program Design, Update JavaPlace* (third edition). Englewood Cliffs, NJ: Prentice Hall, 2002.
10. Richard Rasala, Jeff Raab, and Viera K. Proulx. Java Power Tools: Model software for teaching object-oriented design. *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, February 2001.
11. Stuart Reges. Can C# replace Java in CS1 and CS2? *Proceedings of the Seventh Annual Conference on Innovation and Technology in Computer Science Education SIGCSE Technical Symposium on Computer Science Education*, June 2002.
12. Eric Roberts and Gerald Engel (editors). *Computing Curricula 2001: Final Report of the Joint ACM/IEEE-CS Task Force on Computer Science Education.* Los Alamitos, CA: IEEE Computer Society Press, December 2001. **`http://www.acm.org/sigcse/cc2001/`**.
13. Walter Savitch. *Java: An Introduction to Computer Science and Programming* (third edition). Englewood Cliffs, NJ: Prentice Hall, 2003.
14. Chris Stevenson. Java engagement for teacher training: Proposal for a pilot project to help local secondary computer science teachers develop expertise in Java programming. ACM memorandum, August 2002.