Computer Science Education:
Where Are the Software Engineers of Tomorrow?

By Dr. Robert B.K. Dewar and Dr. Edmond Schonberg

AdaCore Inc.

It is our view that Computer Science (CS) education is neglecting basic skills, in particular in the areas of programming and formal methods. We consider that the general adoption of Java as a first programming language is in part responsible for this decline. We examine briefly the set of programming skills that should be part of every software professional's repertoire.

It is all about programming! Over the last few years we have noticed worrisome trends in CS education. The following represents a summary of those trends:

1. Mathematics requirements in CS programs are shrinking.
2. The development of programming skills in several languages is giving way to cookbook approaches using large libraries and special-purpose packages.
3. The resulting set of skills is insufficient for today's software industry (in particular for safety and security purposes) and, unfortunately, matches well what the outsourcing industry can offer. We are training easily replaceable professionals.

These trends are visible in the latest curriculum recommendations from the Association for Computing Machinery (ACM). Curriculum 2005 does not mention mathematical prerequisites at all, and it mentions only one course in the theory of programming languages [1].

We have seen these developments from both sides: As faculty members at New York University for decades, we have regretted the introduction of Java as a first language of instruction for most computer science majors. We have seen how this choice has weakened the formation of our students, as reflected in their performance in systems and architecture courses. As founders of a company that specializes in Ada programming tools for mission-critical systems, we find it harder to recruit qualified applicants who have the right foundational skills. We want to advocate a more rigorous formation, in which formal methods are introduced early on, and programming languages play a central role in CS education.

Formal Methods and Software Construction

Formal techniques for proving the correctness of programs were an extremely active subject of research 20 years ago. However, the methods (and the hardware) of the time prevented these techniques from becoming widespread, and as a result they are more or less ignored by most CS programs. This is unfortunate because the techniques have evolved to the point that they can be used in large-scale systems and can contribute substantially to the reliability of these systems. A case in point is the use of SPARK in the re-engineering of the ground-based air traffic control system in the United Kingdom (see a description of iFACTS – Interim Future Area Control Tools Support, at <www.nats.co.uk/article/90>). SPARK is a subset of Ada augmented with assertions that allow the designer to prove important properties of a program: termination, absence of run-time exceptions, finite memory usage, etc. [2]. It is obvious that this kind of design and analysis methodology (dubbed Correctness by Construction) will add substantially to the reliability of a system whose design has involved SPARK from the beginning. However, PRAXIS, the company that developed SPARK and which is designing iFACTS, finds it hard to recruit people with the required mathematical competence (and this is present even in the United Kingdom, where formal methods are more widely taught and used than in the United States).

Another formal approach to which CS students need exposure is model checking and linear temporal logic for the design of concurrent systems. For a modern discussion of the topic, which is central to mission-critical software, see [3].

Another area of computer science which we find neglected is the study of floating-point computations. At New York University, a course in numerical methods and floating-point computing used to be required, but this requirement was dropped many years ago, and now very few students take this course. The topic is vital to all scientific and engineering software and is semantically delicate. One would imagine that it would be a required part of all courses in scientific computing, but these often take MatLab to be the universal programming tool and ignore the topic altogether.

The Pitfalls of Java as a First Programming Language

Because of its popularity in the context of Web applications and the ease with which beginners can produce graphical programs, Java has become the most widely used language in introductory programming courses. We consider this to be a misguided attempt to make programming more fun, perhaps in reaction to the drop in CS enrollments that followed the dot-com bust. What we observed at New York University is that the Java programming courses did not prepare our students for the first course in systems, much less for more advanced ones. Students found it hard to write programs that did not have a graphic interface, had no feeling for the relationship between the source program and what the hardware would actually do, and (most damaging) did not understand the semantics of pointers at all, which made the use of C in systems programming very challenging.

Let us propose the following principle: The irresistible beauty of programming consists in the reduction of complex formal processes to a very small set of primitive operations. Java, instead of exposing this beauty, encourages the programmer to approach problem-solving like a plumber in a hardware store: by rummaging through a multitude of drawers (i.e. packages) we will end up finding some gadget (i.e. class) that does roughly what we want. How it does it is not interesting! The result is a student who knows how to put a simple program together, but does not know how to program. A further pitfall of the early use of Java libraries and frameworks is that it is impossible for the student to develop a sense of the run-time cost of what is written because it is extremely hard to know what any method call will eventually execute. A lucid analysis of the problem is presented in [4].

We are seeing some backlash to this

approach. For example, Bjarne Stroustrup reports from Texas A & M University that the industry is showing increasing unhappiness with the results of this approach. Specifically, he notes the following:

> I have had a lot of complaints about that [the use of Java as a first programming language] from industry, specifically from AT&T, IBM, Intel, Bloomberg, NI, Microsoft, Lockheed-Martin, and more. [5]

He noted in a private discussion on this topic, reporting the following:

> It [Texas A&M] did [teach Java as the first language]. Then I started teaching C++ to the electrical engineers and when the EE students started to out-program the CS students, the CS department switched to C++. [5]

It will be interesting to see how many departments follow this trend. At AdaCore, we are certainly aware of many universities that have adopted Ada as a first language because of similar concerns.

A Real Programmer Can Write in Any Language (C, Java, Lisp, Ada)

Software professionals of a certain age will remember the slogan of old-timers from two generations ago when structured programming became the rage: Real programmers can write Fortran in any language. The slogan is a reminder of how thinking habits of programmers are influenced by the first language they learn and how hard it is to shake these habits if you do all your programming in a single language. Conversely, we want to say that a competent programmer is comfortable with a number of different languages and that the programmer must be able to use the mental tools favored by one of them, even when programming in another. For example, the user of an imperative language such as Ada or C++ must be able to write in a functional style, acquired through practice with Lisp and ML[1], when manipulating recursive structures. This is one indication of the importance of learning in-depth a number of different programming languages. What follows summarizes what we think are the critical contributions that well-established languages make to the mental tool-set of real programmers. For example, a real programmer should be able to program inheritance and dynamic dispatching in C, information hiding in Lisp,

tree manipulation libraries in Ada, and garbage collection in anything but Java. The study of a wide variety of languages is, thus, indispensable to the well-rounded programmer.

Why C Matters
C is the low-level language that everyone must know. It can be seen as a portable assembly language, and as such it exposes the underlying machine and forces the student to understand clearly the relationship between software and hardware. Performance analysis is more straightforward, because the cost of every software statement is clear. Finally, compilers (GCC for example) make it easy to examine the generated assembly code, which is an excellent tool for understanding machine language and architecture.

Why C++ Matters
C++ brings to C the fundamental concepts of modern software engineering: encapsulation with classes and namespaces, information hiding through protected and private data and operations, programming by extension through virtual methods and derived classes, etc. C++ also pushes storage management as far as it can go without full-blown garbage collection, with constructors and destructors.

Why Lisp Matters
Every programmer must be comfortable with functional programming and with the important notion of referential transparency. Even though most programmers find imperative programming more intuitive, they must recognize that in many contexts that a functional, stateless style is clear, natural, easy to understand, and efficient to boot.

An additional benefit of the practice of Lisp is that the program is written in what amounts to abstract syntax, namely the internal representation that most compilers use between parsing and code generation. Knowing Lisp is thus an excellent preparation for any software work that involves language processing.

Finally, Lisp (at least in its lean Scheme incarnation) is amenable to a very compact self-definition. Seeing a complete Lisp interpreter written in Lisp is an intellectual revelation that all computer scientists should experience.

Why Java Matters
Despite our comments on Java as a first or only language, we think that Java has an important role to play in CS instruction. We will mention only two aspects of the language that must be part of the real programmer's skill set:

1. An understanding of concurrent programming (for which threads provide a basic low-level model).
2. Reflection, namely the understanding that a program can be instrumented to examine its own state and to determine its own behavior in a dynamically changing environment.

Why Ada Matters
Ada is the language of software engineering par excellence. Even when it is not the language of instruction in programming courses, it is the language chosen to teach courses in software engineering. This is because the notions of strong typing, encapsulation, information hiding, concurrency, generic programming, inheritance, and so on, are embodied in specific features of the language. From our experience and that of our customers, we can say that a real programmer writes Ada in any language. For example, an Ada programmer accustomed to Ada's package model, which strongly separates specification from implementation, will tend to write C in a style where well-commented header files act in somewhat the same way as package specs in Ada. The programmer will include bounds checking and consistency checks when passing mutable structures between subprograms to mimic the strong-typing checks that Ada mandates [6]. She will organize concurrent programs into tasks and protected objects, with well-defined synchronization and communication mechanisms.

The concurrency features of Ada are particularly important in our age of multicore architectures. We find it surprising that these architectures should be presented as a novel challenge to software design when Ada had well-designed mechanisms for writing safe, concurrent software 30 years ago.

Programming Languages Are Not the Whole Story
A well-rounded CS curriculum will include an advanced course in programming languages that covers a wide variety of languages, chosen to broaden the understanding of the programming process, rather than to build a résumé in perceived hot languages. We are somewhat dismayed to see the popularity of scripting languages in introductory programming courses. Such languages (Javascript, PHP, Atlas) are indeed popular tools of today for Web applications. Such languages have all the pedagogical defaults that we ascribe to Java and provide no opportunity to learn algorithms and performance analysis. Their absence of strong typing leads to a trial-

and-error programming style and prevents students from acquiring the discipline of separating design of interfaces from specifications.

However, teaching the right languages alone is not enough. Students need to be exposed to the tools to construct large-scale reliable programs, as we discussed at the start of this article. Topics of relevance are studying formal specification methods and formal proof methodologies, as well as gaining an understanding of how high-reliability code is certified in the real world. When you step into a plane, you are putting your life in the hands of software which had better be totally reliable. As a computer scientist, you should have some knowledge of how this level of reliability is achieved. In this day and age, the fear of terrorist cyber attacks have given a new urgency to the building of software that is not only bug free, but is also immune from malicious attack. Such high-security software relies even more extensively on formal methodologies, and our students need to be prepared for this new world.◆

References

1. Joint Taskforce for Computing Curricula. "Computing Curricula 2005: The Overview Report." ACM/AIS/IEEE, 2005 <www.acm.org/education /curric_vols/CC2005-March06 Final.pdf>.
2. Barnes, John. <u>High Integrity Ada: The Spark Approach</u>. Addison-Wesley, 2003.
3. Ben-Ari, M. <u>Principles of Concurrent and Distributed Programming</u>. 2nd ed. Addison-Wesley, 2006.
4. Mitchell, Nick, Gary Sevitsky, and Harini Srinivasan. "The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications." Workshop on Library-Centric Software Design, Object-Oriented Programming, Systems, Languages, and Applications, San Diego, CA, 2005.
5. Stroustrup, Bjarne. Private communication. Aug. 2007.
6. Holzmann Gerard J. "The Power of Ten – Rules for Developing Safety Critical Code." <u>IEEE Computer</u> June 2006: 93-95.

Note

1. Several programming language and system names have evolved from acronyms whose formal spellings are no longer considered applicable to the current names for which they are readily known. ML, Lisp, GCC, PHP, and SPARK fall under this category.

About the Authors

Robert B.K. Dewar, Ph.D., is president of AdaCore and a professor emeritus of computer science at New York University. He has been involved in the design and implementation of Ada since 1980 as a distinguished reviewer, a member of the Ada Rapporteur group, and the chief architect of Gnu Ada Translator. He was a member of the Algol68 committee and is the designer and implementor of Spitbol. Dewar lectures widely on programming languages, software methodologies, safety and security, and on intellectual property rights. He has a doctorate in chemistry from the University of Chicago.

> **AdaCore**
> **104 Fifth AVE**
> **15th FL**
> **New York, NY 10011**
> **Phone: (212) 620-7300 ext. 100**
> **Fax: (212) 807-0162**
> **E-mail: dewar@adacore.com**

Edmond Schonberg, Ph.D., is vice-president of AdaCore and a professor emeritus of computer science at New York University. He has been involved in the implementation of Ada since 1981. With Robert Dewar and other collaborators, he created the first validated implementation of Ada83, the first prototype compiler for Ada9X, and the first full implementation of Ada2005. Schonberg has a doctorate in physics from the University of Chicago.

> **AdaCore**
> **104 Fifth AVE**
> **15th FL**
> **New York, NY 10011**
> **E-mail: schonberg@adacore.com**