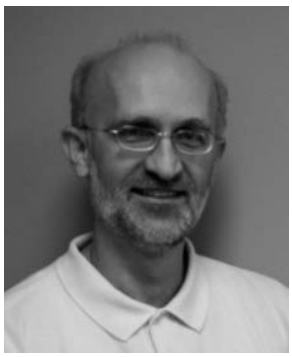
Invited Editorial



Dropping CS Enrollments: Or The Emperor's New Clothes?

Bill Manaris
Computer Science Department
College of Charleston
Charleston, South Carolina USA
manaris@cs.cofc.edu

According to CRA [1], the number of new CS majors dropped by approximately 50% from Fall 2000 to Fall 2006. So, what happened right before Fall 2001 when the CS enrollments started dropping?

There are two anecdotal, yet widely accepted explanations: (a) the "dot-com" bust, and (b) outsourcing. Interestingly, the "dot-com" bubble seemed to be at its highest during 2000 [2], so this is definitely not what initiated the mass exodus of CS students. Could outsourcing alone have this effect? Perhaps.

It's hard to imagine that any single event (e.g., dot-com bust) would immediately affect CS enrollment rates. We need to account for news and word-of-mouth to travel and become part of conventional wisdom (e.g., "you do not want to be a CS major because…").

Another Possibility

Here are two interesting trends in CS education, which started a few years earlier and culminated around Fall 2001 [3]:

- the use of C++ and Java (both industrial-strength OO programming languages) in CS1 increased from 39% (1996-97) to 56% (1997-98) to 72% (1998-99) to 76% (1999-2000) to 89% (2001-02) to a projected 96% (2002-03).
- the adoption of object-oriented as the primary programming paradigm (instead of procedure-oriented) changed from 36% (1995-96) to 55% (1996-97) to 74% (1997-98 and 1998-99) to 83% (1999-2000) to 82% (2001-02).

So here are two questions to consider:

What evidence do we have that introducing OO in CS1 is beneficial, in terms of student retention and performance in later courses?

What evidence do we have that using Java or C++ as the CS1 primary language is beneficial, in terms of student retention and performance in later courses?

As I was pondering these questions, I remembered Jakob Nielsen's thoughts on why usability is important [4]:

> On the Web, usability is a necessary condition for survival. If a website is difficult to use, *people leave*. If the homepage fails to clearly state what a company offers and what users can do on the site, *people leave*. If users get lost on a website, *they leave*. If a website's information is hard to read or doesn't answer users' key questions, *they leave*. Note a pattern here? There's no such thing as a user reading a website manual or otherwise spending much time trying to figure out an interface. There are plenty of other websites available; leaving is the first line of defense when users encounter a difficulty.

As I read this I could not help but think of the potential connection between the choice of CS1 programming language/paradigm and student enrollments/retention.

This is not to say that OO is bad. Nor that Java and C++ are bad languages. However, we need to consider the possibility that they make computer science appear very hard to beginning students.

How Hard Is it to Program?

Alan Kay, in his paper "The Early History of Smalltalk" states [5]:

> When teaching [programming] to 20 nonprogrammer adults, they were able to get through the initial material faster than children, but, just as it looked like an overwhelming success

> was at hand, they started to crash on problems that didn't look to me to be much harder than the ones they had just been doing so well on. … E.g., make a little database system that would act like a card file or rolodex. They couldn't even come close to programming it. Such a project was well below the mythical 'two pages' for end-users we were working with. After showing them the solution, I realized this little program contained 17(!) nonobvious ideas. And some of them were like the concept of the arch in building design: very hard to discover, if you don't already know it.

In this context, here are two typical programs, in Java and in Python, to calculate length of the hypotenuse of a triangle, given the length of the two legs. Why Java and Python? Because both are very successful languages in industry, and both are used to introduce computer science to CS1 students.

Consider the number of new concepts or "nonobvious ideas" required to comprehend/write each of these programs.

Java

For the Java program (see Fig. 1), a programmer needs to master the following "nonobvious ideas":

1. libraries (i.e., import java.util.*;)
2. class
3. encapsulation and information hiding (e.g., public vs. private, etc.)
4. class name must be the same as program file name
5. blocks (i.e., { ... })
6. method
7. special method main() (i.e., program entry point by Java VM)
8. static vs. non-static methods
9. void vs. value returning methods
10. parameter passing
11. arrays (i.e., String[] args)
12. command-line parameter passing (i.e., String[] args)
13. class instantiation (class vs. object)
14. you may instantiate an object within its own class (i.e., chicken-and-egg paradox)
15. statement terminator (i.e., ;)
16. data types
17. primitive data types vs. non-primitive data types
18. variable
19. assignment
20. variable declaration
21. input
22. special class Scanner
23. input streams, e.g., System.in
24. class member method invocation (e.g., input.nextInt())
25. iterators (e.g., nextInt())
26. output (i.e., System.out.println())
27. hierarchical composition (i.e., System constains object out)

```
import java.util.*;

public class Triangle
{
  public static void main (String[] args)
  {
    Triangle t1 = new Triangle();
    Scanner input = new Scanner(System.in);
    int a = input.nextInt();
    int b = input.nextInt();
    double c = t1.hypotenuse(a, b);
    System.out.println(c);
  }

  public double hypotenuse(int a, int b)
  {
   return (Math.sqrt( (a*a) + (b*b) ));
  }
}
```

Figure 1. Pythagorean Theorem in Java.

```
from math import *

def hypotenuse(a, b):
    return(sqrt( (a*a) + (b*b) ))

x = input()
y = input()

print hypotenuse(x, y)
```

Figure 2. Pythagorean Theorem in Python.

28. algebraic expressions
29. Math library (i.e., sqrt())
30. return statement

Python

For the Python[1] program (see Fig. 2), a programmer needs to master the following "nonobvious ideas":

1. module (i.e., from math import *)
2. function
3. blocks (i.e., indentation)
4. statement terminator (i.e., newline)
5. functions may or may not return a value
6. parameter passing
7. variable
8. assignment
9. input (i.e., input())
10. output (i.e., print)
11. algebraic expressions
12. return statement

[1] If you are new to Python, see [6, 7].

The above is a generic, simple (rather typical?) programming task. Given human cognitive limitations (e.g., the 7 plus-or-minus 2 rule [8]), it is not surprising that programmers report at least a 3 times increase in productivity when they move from Java (or C/C++) to Python [9, 10]. It is about *task interference* – how much do you have to focus on the tool (e.g., language syntax, semantics, conceptual model, etc.) vs. the task you are trying to perform (i.e., problem solving).

Also, the above comparison does not capture the *amount of effort* required to master each of the required concepts. Some concepts are easier to master than others (e.g., "statement terminator" vs. "class"). Also, a single concept may be easier to master in one language vs. another (e.g., "input" or "method/function"). For instance, McConnell and Burhans [11] observed that:

> The average size for the Java books in our study is 2.25 times the average size of the Fortran books and 1.95 times the average size of the Pascal books. From the perspective of a course, the authors of three of the four Java books we examined expect that their entire book (averaging 880 pages) will be covered in one semester. This amounts to 22 pages per class or about 66 pages per week that a student is expected to prepare.

It would be interesting to generate a *relative index of difficulty* by, say, counting the number of lines used in a typical CS book to explain each of these concepts, given the syntax, semantics, and conceptual model of each language. These are the same lines of text our students are expected to read, in order to master the concepts.

Usability of Programming Languages

As we are collectively exploring the reasons for our dropping enrollments, it is interesting to note that a programming language is just another user interface (UI). Similar to other UIs such as MS-DOS, Unix, Mac OS X, and Windows, programming languages are an abstraction barrier between the end-user and the underlying machine.

Programming languages, when viewed as user interfaces, may be evaluated formally through usability techniques. According to Jakob Nielsen [4], there are many different attributes for measuring the quality of a user interface, but two key ones are *utility* and *usability*. Utility asks the question "does the UI provide the necessary functionality to achieve your tasks?" Theoretically speaking, if a programming language is *Turing complete*, it has adequate utility for all computable tasks. But most computer scientists would agree that Turing-completeness is not enough.

Usability is a "quality attribute that assesses how easy user interfaces are to use" [4]. If you agree that a programming language can be thought of as a programmer's user interface to a Turing machine, then we may explore its usability in terms of these dimensions:

- *Learnability*: How easy is it to perform basic tasks the first time programmers encounter the programming language?
- *Efficiency*: Once programmers have learned the language, how quickly can they perform typical tasks?
- *Memorability*: When programmers return to the language after a period of not using it, how easily can they reestablish proficiency?
- *Errors*: How many errors do programmers make, how severe are these errors, and how easily can they recover from the errors?
- *Satisfaction*: How pleasant is it to use the language?

According to Nielsen:

> Usability and utility are equally important: It matters little that something is easy if it's not what you want. It's also no good if the system can hypothetically do what you want, but you can't make it happen because the user interface is too difficult.

Examples

How can we go about evaluating the usability of a programming language? Here is a nice example from NASA JPL: http://oodt.jpl.nasa.gov/better-web-app.mov .

In it, the evaluator picks certain tasks (e.g., build a "Hello World" application), and tries to perform them with different programming environments. You watch him work in real time (via screencast), making errors and correcting them, and in the end *reporting quantitative results*. Powerful! Contrast this with the anecdotal diatribes we consume or generate, during language wars, of the type "feature X obviously helps reduce bugs" or "feature Y is better for software design". As a scientist, I vote for the scientific method.

Clearly, in any usability evaluation study, the choice of tasks is essential. You pick the wrong tasks and your results are misleading.

So, the other day, I selected a small task and wrote code both in language A and language B.

Results

With language A, I had to look up two things in the API, had 9 compiler errors, one semantic error, and one "headache" error. (What is a "headache" error? View the above video.)

With language B, I had only one syntax error.

The number-of-lines ratio was 3 to 1 (A to B). Same task.

Before you ask me which languages I used and what the task was, I challenge you to do the same with your A and B, and your task of choice. Try a task from CS1, for example. Just make sure it is not tied to syntax (e.g., inheritance vs. interfaces).

Conclusion

This editorial extends an invitation to the CS education research community to explore the usability of programming languages used in CS education and the effect these languages may have on achieving our goals as educators, including student retention and learning.

I have already shared these ideas with several colleagues, and, to the best of my knowledge, there are at least a couple of CS ed research activities underway related to this theme. But there is so much more to explore.

According to van der Veer and van Vliet [12], in the eyes of the user *the user interface **is** the system*. We should consider the possibility that, in the eyes of our CS1 students, *the programming language/paradigm they are exposed to **is**… Computer Science*.

This is not to say that we should all abandon Java and move to Python or Ruby. This is not to say that Python is the best language for CS1.

Have you considered that our programming user interfaces are still built on 70s user interface technology and concepts (and, even at that, not very well)? To put it bluntly: our modern programming languages are nothing more than "supersized" command-line interfaces. In "Strong Typing vs. Strong Testing" Bruce Eckel suggests [10]:

> It takes an earth-shaking experience - like learning a different kind of language - to cause a re-evaluation of beliefs.

Do you remember the "earth-shaking" experience of using a mouse for the first time? Or, oh my Gosh, the first time you interacted with a graphical user interface?

We are still waiting for our programming languages to catch up and move into the 90s, into the visual domain. Granted there have been several attempts (e.g., see [13] for a thorough overview). However, no visual programming language can currently handle sizable development efforts. However, the potential is there. For example, see *Alice* and *Scratch* – two recent "proposals" on what visual programming languages might look like [14, 15]. Notice how they both support *sequence*, *selection*, and *iteration*. What's missing? See if you can compare them in terms of usability – there are significant differences. Why are they so effective? But perhaps that's a topic for a future op-ed piece.

This article is about the possibility that there is no absolute best language/paradigm for CS1 (the Emperor's New Clothes). Instead, at this period of "lean cows" each department should thoughtfully consider what language/paradigm is best for your students' preparation, abilities, and tasks you expect them to master in CS1. Then perhaps you might discover that your retention rates improve drastically, as anecdotal evidence suggests, and in accordance with what Human-Computer Interaction teaches us about usability and its effects on the effectiveness and popularity of systems.

In the minds of our beginning students, *the programming language/paradigm we expose them to in CS1 **is** computer science*.

References

[1] Vegso, J. "Continued Drop in CS Bachelor's Degree Production and Enrollments as the Number of New Majors Stabilizes", Computing Research News, 19(2), Mar. 2007.

[2] Wikipedia, "Dot-com bubble", accessed Sep. 24, 2007.

[3] McCauley, R. and Manaris, B., Comprehensive Report on the 2001 Survey of Departments Offering CAC -Accredited Degree Programs, May 2002, http://www.cs.cofc.edu/~mccauley/survey.

[4] Jakob Nielsen, "Usability 101: Introduction to Usability", Aug. 2003, http://www.useit.com/alertbox/20030825.html

[5] Kay, A. "The Early History of Smalltalk", *ACM SIGPLAN Notices*, 28(3), March 1993.

[6] Zelle, J.M. "Teaching Computer Science with Python", SIGCSE 2003 Workshop #4 transparencies, accessed Sep. 25, 2007, http://mcsp.wartburg.edu/zelle/python/sigcse-slides.pdf

[7] Hetland, M.D. "Instant Python", accessed Sep. 25, 2007, http://hetland.org/writing/instant-python.html

[8] Miller, G.A. (1956), "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", The Psychological Review, vol. 63, pp. 81-97.

[9] Ferg, S. "Python & Java – A Side-by-Side Comparison", http://www.ferg.org/projects/python_java_side-by-side.html, accessed Sep. 25, 2007.

[10] Eckel, B. "Strong Typing vs. Strong Testing", http://www.mindview.net/WebLog/log-0025, accessed Sep. 25, 2007.

[11] McConnell, J.J. and Burhans, D.T. (2002), "The Evolution of CS1 Textbooks", 32nd ASEE/IEEE Frontiers in Education Conference, November 6-9, 2002, Boston, MA, pp. T4G1-T4G6.

[12] van der Veer, G. and van Vliet, H. (2003), "A Plea for a Poor Man's HCI Component in Software Engineering and Computer Science Curricula; After all: The Human-Computer Interface is the System", *Computer Science Education*, 13(3), pp. 207-225.

[13] Kelleher, C. and Pausch, R. (2005), "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers", *ACM Computing Surveys*, 37(2), Jun. 2005, pp. 83-137.

[14] Stage3 Research Group, "Alice: Free, Easy, Interactive 3D Graphics for the WWW", Carnegie Mellon University, http://www.alice.org/ .

[15] Lifelong Kindergarten Group, "Scratch: Imagine, Program, Share", MIT Media Lab, http://scratch.mit.edu/.

Credits
The Pythagorean Theorem code was written by my students, Brian Smith and Jeff Shumard. My department colleagues contributed through various discussions and thoughtful insights, especially Renée McCauley, Walter Pharr, George Pothering, and James Wilkinson. This work has been supported in part by NSF grant DUE 02-26080.