The Role of Language Paradigms in Teaching Programming

Peter Van Roy (Moder.)
Univ. cath. Louvain
Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be

Joe Armstrong
SICS
Kista, Sweden
joe@sics.se

Matthew Flatt
University of Utah
Utah, USA
mflatt@cs.utah.edu

Boris Magnusson
Lund University
Lund, Sweden
boris@cs.lth.se

Abstract

The purpose of this panel is to confront the wide variety of opinions on the role of language paradigms in teaching programming. We have selected four divergent opinions:

- Armstrong says that concurrent programming is considered difficult because it is taught in the wrong paradigm, namely imperative or object-oriented programming. Instead, concurrency should be taught using a paradigm that makes it simple.
- Flatt says that everyone should be taught how to program, not just computer science majors. Further, programming should be taught as an extension of what students already know, which is algebra. More important than a particular paradigm, however, is teaching students a design process.
- Magnusson says that object-oriented programming must be the first and principal paradigm, because it is best for teaching how to analyze problems and structure solutions. Other paradigms can be taught after students have a solid understanding of OO.
- Van Roy says that programming should be taught in terms of concepts, not paradigms. Common paradigms (functional, OO, etc.) then appear naturally, depending on the concepts used.

The panel will confront these opinions to enrich our understanding of how to teach programming.

Categories & Subject Descriptors

K.3.2 *Computers and Education*: Computer and Information Science Education - computer science education, curriculum. D.3.2 *Programming Languages*: Language Classifications.

General Terms

Languages

Keywords:

Programming Paradigms, Concepts, Concurrency

1 Joe Armstrong

Programs that model or interact with the real world need to reflect the concurrency patterns that are observed in the real world. The real world *is* concurrent—and writing programs to interact with the real world should be a simply a matter of identifying the concurrency in the problem, identifying the message channels and mapping these 1:1 onto the code—the program then almost writes itself.

Unfortunately, concurrent programming has acquired a reputation of being "difficult" and something to be avoided if possible. I believe this is a side-effect of the problems of thread programming in conventional operating systems using languages like Java, C, or C++. In a concurrent language like Erlang, concurrent programming becomes "easy" and becomes the natural way of solving a large class of problems [1].

Most conventional languages that have primitives for concurrent programming provide only a thin layer to whatever mechanisms are offered by the host operating system. Thus Java uses the concurrency mechanisms provided by the underlying operating system and the inefficiency of concurrency in Java is merely a reflection of the fact that the concurrency mechanisms in the operating system are inefficient.

I believe that concurrency should be a property of the programming language and *not* something inherited from the OS. Erlang is such a language. Erlang processes are extremely lightweight: creating a parallel process in Erlang is about 100 times faster than in Java

or C++. That's because concurrency is *designed into the language* and has nothing to do with the host OS.

Once you put concurrency into the language a lot of things look very different—concurrent programing becomes easy. This is especially important in programming high-availability real-time or distributed applications where concurrency is inescapable.

2 Matthew Flatt

Everyone should learn how to program! Programming is not just a specialty discipline limited to computer science majors. Programming is a way of thinking that is useful to everyone. As Alan Perlis has remarked in *Epigrams of Programming*:

> It goes against the grain of modern education to teach children to program. What fun is there in making plans, acquiring discipline in organizing thoughts, devoting attention to detail and learning to be self-critical?

Programming teaches many of the same skills as mathematics and English. Programs have to be constructed with precision. Reading and writing them has to be done with care. But the reward for a correct program is immediate and strong: it executes and gives a result.

The important part of programming is the design process: how problem statements lead to well-organized solutions. Learning this process is like learning to play soccer: the first thing is to learn basic techniques such as trapping a ball, dribbling, passing, and shooting. Only afterwards does the player learn how to play a game.

We have used this approach to teach a first programming course for liberal arts students [2]. We find that very few concepts are needed. We use only six: function definition and application, conditional, local definitions, structure definitions, and destructive assignment. We provide a custom programming environment tailored to these concepts, not a professional environment. More complex concepts, such as objects and classes, are introduced only later when the student is ready for them.

3 Boris Magnusson

Object Orientation is sometimes regarded as an advanced topic that is hard to teach. This might be true if you teach it to students who have a background in another programming paradigm, but our experience is that OO is ideal to use as a first paradigm. This means that objects and classes are introduced from day one and details of expressions, statements, parameters etc., come later as needed.

The order you teach things depends on what is considered important and central in a curriculum. We want our students to understand mechanisms of how to analyze a problem and structure a solution. We thus start with structuring mechanisms, classes, methods, and inheritance so the students get training in the use of these mechanisms throughout the course. Other paradigms, such as procedural, functional, and logic programming, are better taught after a solid understanding of OO.

At our university we have over 10 years experience with this method of introducing programming and our experience supports this view.

4 Peter Van Roy (moderator)

There exist many programming "paradigms": styles of programming based on particular mathematical theories or schools of thought. Some popular paradigms include object-oriented programming, logic programming, and functional programming. Each has its own advantages and disadvantages.

In today's computing curricula, very few programming paradigms are taught. Object-oriented programming dominates by far, with minor attention given to functional or logic programming. Students do not see how the paradigms relate and how they can be used together.

This has a detrimental effect on programmer competence and thus on program quality. For example, students who learn about concurrency only from Java conclude that it is always difficult and expensive. This is false: there are paradigms of concurrent programming that are almost as easy as sequential programming.

One way to solve this problem is to base programming courses on *concepts*, not on single paradigms or languages. We have taught with this approach for two years in four universities, in second-year through graduate courses [3]. We find that it liberates students from the tyranny of single paradigms. Students can reason in a broad and deep way about their program's design, its correctness, and its complexity.

References

[1] Armstrong, J., Williams, M., Wikström, C., and Virding, R. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[2] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. *How to Design Programs: An Introduction to Computing and Programming*. The MIT Press, 2001.

[3] Van Roy, P., and Haridi, S. *Concepts, Techniques, and Models of Computer Programming*. 2002. Draft available at `http://www.info.ucl.ac.be/people/PVR/book.html`.