I consider myself an above-average programmer. Over the past 20 years, I have written more than 100,000 lines of code in many different languages, operating systems, and platforms. I progressed from Basic to Pascal, with a short intermediate stint in assembly language, and by the time I turned 16 I had begun learning C from a third-generation photocopy of The C Programming Language (B. W. Kernighan and D. M. Ritchie, Addison-Wesley, 1978), then vented some misspent creativity on four winning entries at the International Obfuscated C Code Contest. Now, at 34, I find myself humbly trying to improve my coding and design skills by learning new languages and programming approaches. About a month ago, I embarked on a project that first stalled, then spun so badly out of control I doubted that I would walk away from the crash I knew would have to follow.

**TAKEOFF**

It all started when I began developing a refactoring environment for C and C++ programmers, a project that involves parsing and semantically analyzing C and C++ source code. I based my first attempt on the Purdue Compiler Construction Toolset (PCCTS) and a complete C++ parser that John Lilley implemented atop it. I thought that reusing code would save me valuable time and effort—after all, I teach my students that this is so.

However, the PCCTS’s 35,000 lines of code quickly overwhelmed my cognitive abilities. From gleaming the code and its documentation I also learned that parsing and analyzing C++ was a lot more complicated than I expected, due to namespaces, templates, classes, and their interactions. So I also gave up on C++ and decided to concentrate on the much simpler C. I quickly located a yacc grammar and a lexical analyzer that Jutta Degener, another above-average programmer, wrote in 1993.

Within a couple of hours, I could parse some simple programs. Having a simple grammar, I now needed a C preprocessor. Here I must clarify that I could not build on the code base of the GNU C compiler because of the licensing restrictions it places on work derived from its source code. The Decus—DEC, now Compaq, code defined only two different C structures. In addition, many aspects of its operation were limited by fixed-size buffers based on the minimum limits dictated by the draft C standard—I can now appreciate why standards specify such limits.

**A BUMPY FLIGHT**

At the time, I persisted with the Decus code due to the difficulty of reimplementing the C preprocessor—a task that proved to be increasingly daunting as I examined the code that performed the macro expansion. A few days later, I had introduced enough of the ANSI C functionality to preprocess the entire 31,000 nonempty lines of the Microsoft Windows SDK windows.h header file, and the files it includes, into a result almost identical to what the Microsoft compiler produced. Only then did I realize that the lack of abstraction I had considered quaint while adapting the source code to handle ANSI C would make any further progress toward integrating the preprocessor with the rest of my system practically impossible.

**Drawing on airline practices, the author proposes coding in pairs, with a senior programmer guiding a junior counterpart.**

Spending a few more hours with the PCCTS source code convinced me that it was not a viable solution either. PCCTS has classes for abstracting everything except the specific functionality I needed to modify. Returning to the drawing board, I decided to implement the C preprocessor from scratch. Naturally, I would do it right: Although I do not consider myself experienced in object-oriented design nor an expert C++ programmer, I saw that I could abstract the various preprocessing phases—trigraph substitution, newline elimination, tokenization, command processing, and macro expansion—as separate classes linked together into a serial stream. Moreover, to backtrack all
these classes I would need to fetch and push back lexical items into their upstream link. I could therefore embed that common functionality into a superclass. Being a cautious type, I decided to completely implement a single class and thereby test my design’s principles.

That’s when things started to get ugly. The return type of the subclasses would not match that of the superclass. Nor could the class I intended to use as a stack for frozen included files handle open files gracefully. At the same time, I began to feel uneasy because I was reimplementing functionality already available, albeit with additional baggage, as part of the C++ standard template library.

ENGINE TROUBLE

I felt helplessly mired in a hopeless situation. I browsed a couple of design pattern books in my library, but found nothing concretely related to my problem. I could not ask any of my peers for help as most had given up programming long ago to pursue managerial positions or academic interests. Although I know many excellent programmers, most would be too busy to help me in such a task. I also felt that those who did have time would not have enough experience with the problems I faced.

Could it be that the problems confronting me derived from the new problem domain and, by adopting a trial-and-error approach, I could quickly overcome them? I didn’t think so. When programming in Pascal, C, Prolog, or even assembly, I could gradually improve my programming style by reading and learning from code others had written. I could also improve my older programs piecemeal by introducing new techniques I had mastered, such as the use of structures, dynamic memory, abstraction, recursion, and coding with type-safety and portability in mind.

All these activities required a relatively small learning investment and provided immediate feedback on the suitability of a particular approach. In this project, however, I faced hard design choices, brittle classes, and a feedback cycle that would span weeks of hard work. The prospect left me terrified.

HELP FROM ABOVE?

In desperation, I recalled my previous academic appointment: It involved a weekly commute between Athens and Samos in the Aegean archipelago. As the pilots negotiated the tricky Samos airport approach, they would deftly bank the aircraft toward the airstrip, avoiding the steep mountains that lay before it. Sometimes, they would target their final approach to the right of the runway, letting the force 7 meltemi wind push them toward the intended landing position. From the open cockpit door I could sometimes see the view from its window rapidly change between sky, land, and sea in response to the aircraft’s nervous dance, while a pilot—often someone younger than me—expertly handled the dozens of buttons, knobs, levers, and dials, safely bringing the huge airplane to a stop.

During these flights, I would often reflect that the pilot might be making his initial flight with passengers aboard or was perhaps making the difficult approach to the Samos airport for the first time. However, I was never worried because, always, beside the young pilot sat a much older copilot, one with hundreds of hours of flight experience. This veteran constantly monitored the approach, guiding his younger apprentice yet ever ready to take the controls should a serious problem arise.

SAFETY PROCEDURES

The airline industry has, over many years, developed and refined procedures and processes to establish a formidable
EXPERIENCE IS MY COPilot

Kent Beck, in his insightful book *eXtreme Programming Explained* (Addison-Wesley, 2000), espouses pair programming and describes the many advantages that derive from having programmers alternately code and help oversee the coding with advice and design assistance. Notice, however, that I never flew to Samos with two 24-year-old pilots at the controls. Inherent in the pilot-and-copilot approach are experience, formalized training, and seniority. Many software engineering researchers and practitioners have expressed dismay with the natural career evolution of experienced programmers, which forces them to stop programming and manage new, inexperienced programmers.

Imagine instead how our profession would evolve if every programmer always coprogrammed with a more experienced, senior counterpart.

To avoid having this system degenerate into a travesty in which a seasoned Cobol programmer attempts to oversee a young Turk coding in Java, we must borrow an additional item from the airline industry: type certification. Organizations that choose to adopt this approach should ensure that both members of the programming pair are familiar at a basic level with the technology they’re using, be it SQL, C, Java, or Cobol. Further, the senior member should have considerable experience with that technology.

SMOOTHING A ROUGH LANDING

How would such an approach have benefited my current plight? First, at age 34, I would still be able to count on the advice of a senior and experienced programmer sitting by my side. Only the fields of professional sports and software engineering consider 30-year-olds to be senior members.

Second, over the past 10 years I would have received valuable practical mentoring in programming, instead of trying to filter the wheat from the chaff only by reading professional magazines, journals, books, and source code. Finally, I would also be able to discuss my problems with my peers who, looking forward to a financially and professionally rewarding career as senior programmers, would hopefully still be programming.

Does such an approach make economic sense? It could. Given that there is a documented 10-to-1 difference between the productivity of the best and worst programmers (H. Sackman, W.J. Erikson, and E.E. Grant, “Exploratory Experimental Studies Comparing Online and Offline Programming Performance,” *Comm. ACM*, Jan. 1968, pp. 3-11), any approach that can cultivate or retain the right sort of talent will make a positive difference in a company’s bottom line. I also believe that 15 years of intense mentoring will yield professionals who occupy the top side of the productivity curve. Further, these professionals will be much more likely to continue working as senior programmers if such a career path is open to them.

S

etting up a system that pairs junior programmers with senior mentors presents a nontrivial challenge. To meet it, we will need globally recognized standards for certification and programming experience and an official and portable way to measure, recognize, and remunerate “programming time”—our industry’s equivalent of airline flight time. We will also need to overcome the software industry’s natural suspicion of a practice that appears to increase the workforce and salary pressures. Establishing such a system, however, will be a sure sign our profession has matured.

Diononis Spinellis is an assistant professor in the Department of Management Science and Technology at the Athens University of Economics and Business. Contact him at dds@aeub.gr.

Neville Holmes, School of Computing, University of Tasmania, Locked Bag 1-359, Launceston 7250; neville.holmes@utas.edu.au