JOURNAL OF COMMERCE & TRADE

Vol. VI No. 2 / October 2011
ISSN: 0973-4503 RNI : UPENG 2006/17831

Pp. 27-30

Software Engineering in the Academy

AUTHOR

Lokesh Khurana
S.P.C. Degree College, Baghpat.

<<< Abstract

There is no universally accepted definition of software engineering. For some, software engineering is just a glorified name for programming. If you are a programmer, you might put "software engineer" on your business card but never "programmer." Others have higher expectations. A textbook definition of the term might read something like this: "the body of methods, tools, and techniques intended to produce quality software." Rather than just emphasizing quality, we could distinguish software engineering from programming by its industrial nature, leading to another definition: "the development of possibly large systems intended for use in production environments, over a possibly long period, worked on by possibly many people, and possibly undergoing many changes," where "development" includes management, maintenance, validation, documentation, and so forth.

1. INTRODUCTION

Judging by the employment situation, current and future graduates can be happy with their choice of studies. The Information Technology Association of America estimated in April 2002 that 850,000 IT jobs would go unfilled in the next 12 months. The dearth of qualified personnel is just as perceptible in Europe and Australia. Salaries are excellent. Project leaders wake up at night worrying about headhunters hiring away some of their best developers—or pondering the latest offers they received them. Although this trend shows no sign of abating in the near future, we should not take the situation for granted. An economic downturn can make employers more choosy. In addition, more people are learning how to do some programming, aided by the growing sophistication of development tools for the mass market. It is likely, for example, that many of the estimated six million people who are Visual Basic developers have not received a formal computer science education. This creates competition and will force the real professionals to stand out. In fact, talking to managers in industry reveals that they are not just looking for employees—they are looking for excellent developers. This is the really scarce resource. The software engineering literature confirms3 that ratios of 20 are not uncommon between the quality of the work of the best and worst developers in a project; managers and those who make hiring decisions soon learn to recognize where a candidate fits into this spectrum. The aim of a top educational program is to train people who will belong to the top tier. Reflecting on why life has been so good, we may note that our constituency—the people who commission and use our systems—has been remarkably tolerant of our deficiencies.

2. PRINCIPLES

Among the most important things that professional software engineers know are concepts that recur throughout their work. Most of these concepts are not specific techniques. If they include a technique, they go beyond it to encompass a mode of reasoning. This defines the most exciting aspect of being a professional software engineer: the mastery of some of these powerful and elegant intellectual schemes that, more than any particular trick of the trade constitute our profession's common treasure. Most of them cannot be taught in one sitting but rather are learned little by little through trial, error, and skillful mentoring. The sidebar "The Principles: What Software Professionals Know" characterizes a few of these concepts.

3. PRACTICES

At a more mundane level, teaching software engineering also involves making the students familiar with practical techniques that have proved to be productive and are a key part of the trade. Examples include the following:

- *Configuration management*. Although it is one of the most important practices that every project should apply, configuration management is not used as widely or systematically as it should be. Configuration management is based on simple principles and supported by readily available tools.
- *Project management.* It does not always have to be such a hard task, but many software engineers are terrible at project management. Although the ample literature on software project management is not perfect, it contains gems that should be taught to all software students because most of them will at some point exert a project management role.
- *Metrics*. This is one of the most underused techniques in software development. Much of the current literature on metrics is not very good because it lacks a scientifically sound theory of what is being measured and why it is relevant. All the same, we should teach students how to use metrics to quantify applicable project and product attributes, to evaluate the claims of methods and tools through objective criteria, and to use quantitative tools as an aid to prediction and assessment.
- *Ergonomics and user interfaces.* Users of software systems expect high-quality user interfaces; like the rest of the system, the user interface must be engineered properly, a skill that can be learned.
- *Documentation.* Software engineers do not just produce software—they should also document it. A course on technical writing should be part of any software curriculum. Here engineering meets the humanities.
- *User interaction.* The best technology is useless unless it meets the needs of its intended users. A good software engineer must know how to listen to customers and users.
- *High-level system analysis.* To solve a problem through software, you must first understand and describe the problem. This task of analysis is an integral part of software engineering, and it's as difficult as anything else in it.
- *Debugging.* Errors and imperfections are an integral part of the software engineer's daily work. We need systematic and effective debugging techniques to cope with them. It is not hard to find other examples of strong, robust techniques that every professional should know and practice.

4. APPLICATION

Under the heading "applications," I include the traditional specific areas of software techniques: fundamental algorithms and data structures, compiler writing, operating systems, databases, artificial intelligence techniques, and numerical computing. The aim here is not to be imperialistic by attaching these disciplines artificially to software engineering. On the contrary, it is to insist that whatever their individual traditions, techniques, and results may be, these are software subjects, and we should teach them in a way that is compatible with the particular view of software engineering the institution chooses. The advantage is mutual: The specialized subjects benefit from more methodologically aware students—for example, programming projects can

focus on the subject at hand, rather than being distorted by pure programming issues because the students have already learned general design and programming skills—and they help meet software engineering goals by providing a wealth of new examples and applications.

5. TOOLS

The fashionable tools of the moment should not determine pedagogy. Indeed, Parnas has some rather strong words to say against teaching specific languages and tools. But if these aspects should not be at the center, we also should not ignore or neglect them. We must expose students to some of the state-of-the-art tools that industry uses. This exposure should proceed with a critical spirit, encouraging students to see the benefits and limitations of these tools—and to think of better solutions. A tools curriculum cannot and should not be exhaustive; it is better to select a handful of programming languages and a few popular products and help the students understand them in depth. If they need other tools, they will learn them on the job. But they must have seen a few during their studies to have a general idea of what's available and what their future employers expect.

7. THE INVERTED CURRICULUM

An idea that complements the multiyear project is to capitalize on one of the great promises of modern software technology: reuse. The principle of the inverted curriculum (a term borrowed from debates on electrical engineering education8), or "progressive opening of the black boxes" (a somewhat longer name but more precise9), is that the students first use powerful tools and components as clients for their own applications, and then progressively lift the hood to see how things are made, make a few modifications, and add their own extensions. The progression is from the consumer side to the producer side, but focuses from the start on powerful and possibly large examples. There are several benefits. Right from the beginning, the students get to deal with impressive programs, like those that handle graphics. The teaching capitalizes on this "wow effect" and the ability to work with immediately visible results. Today's students have used electronic games and PCs from an early age, and they will not be too impressed by the typical introductory programming examples (the eight queens and such). Trying to get them excited is pedagogy, not demagogy.

8. FUTURE SCOPE

Wirth's compilers made the Swiss Federal Institute of Technology in Zurich a household name in the software world, but it is hard to imagine a compiler, however innovative, achieving a similar result today. A university group would have a tough time competing with the hundreds of developers behind Microsoft's Visual C++ or even those behind the GNU GCC compiler—not a commercial effort but also not an academic one in any accurate sense of the term. People, students included, expect a compiler to come with a sophisticated development environment with all the trappings—a visual debugger, browser, graphical user interface designer, and configuration management facilities. For all the criticism that academic circles give Visual C++ and similar tools, which they may deserve in part, they provide a wealth of resources and facilities (some, it must be said, very cleverly devised), setting a high standard for anyone who wants to compete. If such competition is hard to sustain nowadays in former areas of academic excellence, academics must find new markets in which they can make their mark. I make no pretense of knowing what all these new fields will be, but one that I find promising is the convergence of component-based development and quality. The industry claims that it is widely embracing the notion of reusable components. But there is no guarantee of quality for these components, no standard, no rules, and no qualifying agency. The risks are as huge as the opportunities. A major endeavor can fail in a catastrophic way because of a small deficiency in one of its more humble components. This kind of situation led to the failure of the initial launch of the Ariane 5 rocket—due to the poorly executed reuse of a minor software component—and delayed the entire industrial enterprise by a year and a half, costing European taxpayers an estimated $10 billion.10 Quality, however, is—or should be academia's specialty. Huge opportunities can spring from this

convergence. A long-term project, the source of PhDs, papers, industry collaborations, and a robust reputation, might involve

- Defining standards for components;
- Developing model high-quality components for everyone to appreciate, criticize, and emulate;
- Setting up qualification metrics, possibly a component maturity model;
- Setting up qualification suites;
- Developing new methods and tools for better components, including proof technology, testing techniques, documentation techniques, and validation techniques; and
- Setting up an organization to qualify and label components that third parties submit.

9. CONCLUSION

For all the talk about "software engineering" in the literature, this article included, we must accept that the term remains in part a slogan, as it was when first introduced almost 35 years ago. Since then, however, we have learned enough to teach our students a coherent set of principles and techniques, without hiding from them or ourselves the many remaining uncertainties. I have tried to maintain a balance here between the conceptual and the operational the principles *and* techniques as I think a software curriculum should do. I have tried to show that we do not need to sacrifice either of these aspects for the other, and to describe a challenge worth tackling: to set up a program of teaching and research that is at the same time serious, ambitious, attractive to the students, technically up to date, firmly rooted in the field's practice, and scientifically exciting.

REFERENCES

1. *D.L. Parnas, "Software Engineering Programmes Are Not Computer Science Programmes,"* CRL Report 361, *Communication Research Laboratory, McMaster Univ., Apr. 1998; to be published in* Annals of Software Eng., *2001.*
2. *Information Technology of America, "Major New Study Finds Enormous Demand for IT Workers: Research Pinpoints Hot Jobs and Skills Needed, Offers Insights on Employer Preferred Training Approaches," http://www.itaa.org/news/pr/PressRelease.cfm?ReleaseID=955379119.*
3.. *B.W. Boehm,* Software Engineering Economics*, Prentice Hall, Upper Saddle River, N.J., 1981.*
4. *P.G. Neumann, "Illustrative Risks to the Public in the Use of Computer Systems and Related Technology," http://www.csl.sri.com/users/neumann/illustrative.html.*
5. *D. Tsichritzis, "The Changing Art of Computer Science Research," in* Electronic Commerce Objects, *D. Tsichritzis, ed., Centre Universitaire d'Informatique, Université de Genève, 1998.*
6. *B. Meyer,* Introduction to the Theory of ProgrammingLanguages*, Prentice Hall, Upper Saddle River, N.J., 1990.*
7. *C. Mingins et al., "How We Teach Software Engineering,"* J. Object-Oriented Programming, *Feb. 1999, pp. 64-75.*
8. *B. Cohen, "The Education of the Information Systems Engineer,"* Electronics & Power*, Mar. 1987, pp. 203-205.*
9. *B. Meyer,* Object-Oriented Software Construction, *2nd ed., Prentice Hall, Upper Saddle River, N.J., 1997.*
10. *J. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane,"* Computer*, Jan. 1997, pp. 129-130.* Bertrand Meyer is chief technology officer of Interactive Software Engineering, Santa Barbara, Calif.,and an adjunct professor at Monash University, Melbourne.His books include *Object-Oriented Software Construction* (Prentice Hall, Upper Saddle River, N.J.,1997). Contact him at Bertrand_Meyer@eiffel.com.