Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 74 (2005) 109–111
The Journal of Systems and Software
www.elsevier.com/locate/jss

Guest editorial

The new context for software engineering education and training

1. Introduction

The ''software crisis'' of the last decade was characterized as one of low programmer productivity and poor software quality Gibbs, 1994. This description was a shot across the bow for computer science educators, who shared the blame for the fact that their graduates were performing poorly. Then arose a severe shortage of software professionals—perhaps partly because of their low productivity and the need to fix their poor quality products (e.g., Y2K problems). Traditional computer science departments were also unable to graduate enough software engineers to meet the seemingly insatiable demand. Could not computer science and/or software engineering educators do better? The creation of many software engineering programs (at both undergraduate and graduate levels) and proposals for many model curricula have provided substantial help. There is no doubt that software engineering and software engineering education are evolving into a true engineering discipline. There are many indications of this new direction: CCSE efforts, SWEBOK initiative, the IEEE/ACM codes of ethic, and the Conference on Software Engineering Education and Training (CSEE&T) which will be held for its 17 year in 2004.

Many software engineering problems have faded, or perhaps mutated, during the past few years—only to be replaced by new ones that seem at least equally challenging for the software engineering education and training community. Software engineering has been punctuated by rapid changes in both the technical and politico-economic conditions under which it is practiced. On the technical side, there have been changes in the computing platform. Today's software engineers must know how to deal with applications built from software components that are distributed over the web. The basic conceptual model of how such software operates is, frankly, quite convoluted and therefore more likely to detract from than to improve overall software quality. But this model is a *fait accompli*. The consequence is that a modern software engineering education should cover technical concepts that most currently practicing software engineers (like most software-engineering educators) never learned in school.

Meanwhile, industry has found its own solution to the shortage of software engineers in the developed world. It has expanded dramatically the outsourcing of some software engineering activities to the developing world—sometimes to companies half way around the globe whose employees speak different languages and who work during times that are several hours out of phase with those of the home office. Today's software engineers must understand how to communicate precisely about software requirements and related details in a truly global environment. They must know how to participate in, to coordinate, to manage, projects on which work is being done literally around the clock and around the world.

It is little wonder that software engineering education and training is an exciting area in which to work. For in addition to the above challenges, the debates continue about what software engineering is, whether it is an engineering discipline or what would make it so, whether there should be undergraduate as well as graduate degree programs in software engineering (different from those in computer science), whether software engineers should be licensed/certified, etc. (Bagert et al., 2001; Henderson, 2003; Saiedian et al., 2002). All of these questions are important. Still, the central issues in software engineering education and training concern the same two questions as always:

- *Content:* What specific ideas should be taught in software engineering courses and curricula? The rapid pace of technological change, and especially the total change that a software engineer will experience over a career of 30–35 years, suggests that content should focus significantly on technology-independent principles underlying the design and construction of large software-based systems. It is safe to say that there is no general agreement about what all these principles are; but also that some such principles are widely if not universally accepted and that students can start to learn them as early as CS1. At the same time, it is important to teach some software technologies,

0164-1212/$ - see front matter
doi:10.1016/j.jss.2003.09.023

and to expose students to currently popular methodologies through which time-tested principles are applied in practice. One big problem for educators is that there are so many of these to choose from! Another problem is that the jury is still out on how well most of these methodologies work in practice, let alone in the classroom.

- *Pedagogy:* What are the best ways to teach the ideas that are selected? Evidence clearly shows that students learn best by doing, by practicing, and that projects are therefore always an important feature of software engineering courses. Yet, technological advances have raised new pedagogical questions along with content questions. In particular, the availability of the web and concomitant new possibilities for distance education have led to many investigations of combining tried-and-true teaching techniques with new methods of delivery to physically distributed audiences.

It is in this dynamic context that we received 52 paper submissions for the Software Engineering Education and Training track of the 25th International Conference of Software Engineering, which was held in May 2003 in Portland, Oregon, US. Each was reviewed by at least three program committee members, and 11 papers were accepted for and presented at the conference. This special issue includes extended versions of three of them, along with an invited additional paper that is especially relevant.

2. Contributions to the special issue

As software engineering degree programs emerge, the need for tools and methodologies that can be effectively used throughout the curriculum also emerges. Bagert and Mengel discuss a project process which was developed at Texas Tech University and has been used successfully in their undergraduate and graduate software engineering courses there over the last five years. Thus, the project process can be introduced in early software engineering classes, and the students become well-versed in its use by the capstone experience courses. The process is tailored for use in various classes and types of projects.

The process is entirely web-based (including templates), which helps to facilitate communication with all stakeholders across platforms; this is especially useful for distance education and when the project client is not locally based.

An existence proof for the effective portability of the process is the successful transfer of it to the software engineering courses at the Rose-Hulman Institute of Technology, which has a school and program structure somewhat different from that of Texas Tech.

The next paper, by Jean-Guy Schneider and Lorraine Johnston from Swinburne University of Technology, Melbourne, Australia, addresses the issue of whether eXtreme Programming as it stands today should be introduced for educating undergraduate software engineering students. Their study is motivated by the fact that "traditional" software engineering curricula in tertiary education are perceived as being very document-centric and do not introduce students enough to "hands-on" practices used in industry.

In their paper, they first define a number of educational objectives for undergraduate software engineering courses and then evaluate the practices of eXtreme Programming against these objectives. Novel to their study is that they do not investigate individual practices such as pair programming in isolation, but instead analyze their impact from a "holistic" perspective, i.e. they consider eXtreme Programming as a *package*. They examine which of the initially set objectives can be met using eXtreme Programming given the constraints of a tertiary teaching environment and which ones cannot. In particular, they suggest that students require more maturity to comprehend all aspects of software engineering, especially whether or not certain quality assurance procedures could be omitted.

They conclude that while individual practices of eXtreme Programming may be helpful for educating students about small scale development, as a package it does not provide students with a wide enough experience to fully understand the importance of *engineering* in software development.

The third paper is by Hedin, Bendix, and Magnusson from Department of Computer Science at Lund University, Sweden. They have used Extreme Programming (XP) with success to substitute for a more traditional waterfall model on a project that introduces team-programming aspects of software engineering to a large group of students.

They have scaled down projects to 14 hours of weekly work for seven weeks, doing six iterations and a final peer evaluation. Still projects expose students to most problems and all phases of software development by a team. The six iterations provides students with six occasions for reflection, learning and self-planned improvement. Projects are done "by-the-book" applying all XP practices with more focus on quality, process and learning than on the product.

They find that it is fundamental to their success that they provide proper support for the projects. This includes a short course on XP practices and labs on the tools to be used, but—more importantly—also dedicated on-site coaches, team in one room, and on-site customer. Still it is not a drain on teaching resources as coaching is done by older students following a special coaching course. In exchange for their work they get deeper knowledge and practice of XP and software

architecture, team management knowledge and experience, and study credits.

Finally, Perry Alexander presents a discussion and postmortem analysis of a software engineering course that integrated formal methods with traditional techniques. Z was taught in the lecture course with an emphasis on defining requirements for software components. In the laboratory, traditional object-oriented techniques were taught with an emphasis on integrating formal with traditional techniques. A capstone project required the students to understand where formal techniques fit in the traditional development cycle and where they complement traditional techniques. The course was offered to undergraduate and graduate students for six years with surprisingly successful results. Alexander's paper is actually an invited paper.

Acknowledgements

We thank the organizers of the conference, and especially the Software Engineering Education subcommittee, the select few others who helped with their reviews, the ICSE Program Chairs, the editorial office of the *Journal of Systems and Software*, and of course the authors for all their efforts which made the production of this special issue of the *JSS* possible, and hope you enjoy their collective contributions.

References

Bagert, D., Dupuis, R., Freeman, P., Saiedian, H., Shaw, M., Thompson, J., 2001. Software engineering body of knowledge, Proceedings of the 23rd International Conference on Software Engineering (ICSE'01). Toronto, Canada, May 2001, pp. 693–696.

Gibbs, W., 1994. Software's chronic crisis. Scientific American 271 (3), 86–95.

Henderson, P., 2003. Software engineering education (SEEd). ACM Software Engineering Notes 28 (4), 3–5.

Saiedian, H., Bagert, D., Mead, N., 2002. Software engineering programs: dispelling myths and misconceptions. IEEE Software 19 (5), 35–41.

Hossein Saiedian (Ph.D., Kansas State University, 1989) is currently a professor of Software Engineering in the Department of Electrical Engineering and Computer Science at the University of Kansas. Professor Saiedian's primary area of research is software engineering and in particular models for quality software development, both technical and managerial ones. He has over 100 publications in a variety of topics in software engineering and computer science.

Bruce W. Weide is a professor of Computer and Information Science at The Ohio State University, where he co-directs the Reusable Software Research Group. His interests include all aspects of software component engineering, especially in applying RSRG work to practice and in teaching its principles to beginning CS students. He and colleague Tim Long were awarded the IEEE Computer Society's 2000 Computer Science and Engineering Undergraduate Teaching Award for their work in the latter area. Weide holds a Ph.D. in Computer Science from Carnegie Mellon University and a B.S.E.E. from the University of Toledo. He is a member of the IEEE, ACM, and CPSR.

Hossein Saiedian
Electrical Engineering and Computer Science
The University of Kansas
Lawrence, KS 66045, USA
E-mail address: saiedian@eecs.ku.edu

Bruce W. Weide
Department of Computer Information Science
The Ohio State University
Columbus, Ohio 43210, USA
E-mail address: weide@cis.ohio-state.edu

Available online 5 January 2004