The Journal of Systems and Software 147 (2019) 104–105

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Editorial

Advancing software engineering education: New practices and perspectives

Welcome to the special issue of the *Journal of Systems and Software* (*JSS*) on software engineering education (SEE). SEE is a discipline that focuses exclusively on the study of how to effectively teach software engineering. SEE has become an independent and mature field. The first SEE conference was held more than 30 years ago. The software engineering field has matured and advanced in many ways. Because of the complexity of the field, we can no longer restrict our curriculum, course delivery, and instructional methods to the practices developed years ago. Many of the older SE problem solving methods have faded over the years and many new ones, that are equally or more challenging, have emerged. As educators, we have to devise new methods of course delivery, adjust our teaching methods, expand our tools and methods, and be aware of the technological advances and best practices that industrial practitioners and leaders promote.

The Conference on Software Engineering Education and Training (CSEE&T) has been providing a forum for the researchers and educators to present and debate new perspectives and progresses in SEE.

The 30th CSEE&T was held in Savannah, Georgia, during 7–9 November, 2017. It included a number of paper sessions, panels, and workshops and provided an excellent forum to discuss some of the very relevant topics related to content delivery, innovation, and instructional approaches to software engineering education, and best practices in software engineering training and university-industrial collaboration and partnership. The major themes of the conference, organized in terms of paper sessions, included: course design, DevOps and mobile development, project-based learning, modeling, online programming, agile processes, learning strategies, online and novice education, and human aspects. In addition to the paper sessions, three very relevant panels attracted and entertained the conference attendees. These included a panel entitled “How to Enhance Diversity in Software Engineering Programs?" (led by Hossein Saiedian, Grace L. Lewis and Andrew B. Williams), one entitled “Undergraduate Software Engineering Programs,” (led by Chris Taylor, Kevin Gary, James Kiper, Carol Wellington, Norha M. Villegas, and Lily Chang), and one entitled “A CSEE&T 2017 Panel Session,” (led by Rich Hilliard). The conference also included a workshop, “Crafting the Future of Software Engineering Education in CC2020: A Workshop,” (instructed by Richard J. LeBlanc, Nancy R. Mead, John Impagliazzo).

The current special issue of the *JSS* was motivated by the conference. We considered the best papers of the conference but also issued an open call for papers related to the theme of the conference. The authors of the best papers were asked to extend their papers to make it sufficiently different from the conference and include additional empirical data. The extended papers were reviewed by the original reviewers but also one or two new reviewers. The following is a summery of the accepted papers.

An Environment with Static and Dynamic Visualization

Jeong Yang, Young Lee, and Kai H. Chang discuss “Evaluations of JaguarCode: A Web-Based Object-Oriented Programming Environment with Static and Dynamic Visualization.” To increase program comprehension and overcome learning obstacles of Object-Oriented Programming (OOP), this research introduces a web-based programming environment, JaguarCode, which supports Java programming along with UML diagrams (class, object, and sequence) and execution traces of programs. JaguarCode uses an approach to integrate the structural and behavioral aspects of OOP in a platform-independent environment. It provides a synchronized static and dynamic visualization of Java programs at line level and a full overview of a project under development. It aims to help students better understand the static structure and dynamic behavior of the programs, as well as object-oriented design concepts. Their paper reports on the evaluation results of JaguarCode regarding its effectiveness and user satisfaction through quantitative and qualitative experiments. The quantitative evaluation study explored differences in correctness and time usage on program understanding problems. The results of the experiments support the conclusion that students in the experimental group using visualizations performed better to answer questions correctly than the controlled group. The application of t-tests rejects the null hypothesis that the correctness is significantly increased by the availability of visualizations in JaguarCode. About the response time, the statistical analysis reveals that, for the relatively hard project, there is a significant difference between the controlled and experimental groups. There was also an interesting finding of how both visualizations did affect students' understanding of program execution. Students took longer to answer, in particular, the relatively difficult questions using the visualizations provided in JaguarCode, which led to higher accuracy in answering the questions correctly. In the qualitative evaluation, student feedback on the usability of JaguarCode interface was studied whether the visualizations help students make their OOP learning easier, their understanding of OO concepts better, and whether the interface would contribute to giving satisfaction. The results of the qualitative evaluation

https://doi.org/10.1016/j.jss.2018.09.023
0164-1212/© 2018 Published by Elsevier Inc.

support the positive effect of JaguarCode on helping students understand OO concepts and meeting goals of providing comfortability and satisfaction for the observations of visualization and usability.

Applying a Maturity Model during a Software Engineering Course

Andreas Bollin, Elisa Reci, Csaba Szabo, Veronika Szaboova, and Rudolf Siebenhofer discuss "Applying a Maturity Model during a Software Engineering Course - Experiences and Recommendations." In industry, the benefit of maturity models is uncontested, and models like CMMI are normally taught in at least advanced software engineering courses. However, when not being part of real-world projects, the added values are difficult to be experienced on first hand by their students. On the other side, teaching itself can be seen as a process (or even better: as a set of related processes) involving the educators, the environments and the learners. So, key ideas stemming from the field of maturity models (e.g. measurement steps, quality improvement, and generic/specific practices to be followed) could be applied to it, too - providing a lot of opportunities for improving the lectures and for showing the usefulness of such models. The authors report on a study and teaching approach where, in three successive semesters and at two different institutions, they started rating the process-maturity of students solving tasks in their software engineering courses and transparently related the maturity levels to the task performances. To do so, they defined their own teaching process and, for one process area, they selected five dimensions (planning, setup, task solving speed, team discussion, and mood) to be observed. Next, they looked for associations between these dimensions, the overall course performance, and the quality (in the sense of completeness and accuracy) of the task preparation of the students. It turned out that, apart from speed, medium-sized correlations between most of the measures and the course performance exist. It also turned out that the quality of the project plans (that were to be prepared by the students before starting to work on their tasks) had a non-negligible influence on the overall performance, but that this factor is only weakly related to process maturity. Moreover, the authors noticed that their students were surprised to experience the use(-fulness) of a maturity model at first hand. They were eager to make use of the practices in their ongoing course works, yielding better results at the end. For that reason, even though their teaching maturity model is at the very beginning, they recommend to transparently introduce an "observation reflection improvement" cycle in any (software engineering) lecture and to start measuring the above mentioned dimensions. Such an approach might yield to students' process-improvement steps during the course, help in fostering the understanding of the term process maturity, and, finally, also might help in improving the overall students' course performances.

Undergraduate Requirements Engineering Course

Chandan R. Rupakheti, Mark Hays, Sriram Mohan, Stephen Chenoweth, and Amanda Stouder present "On a Pursuit for Perfecting an Undergraduate Requirements Engineering Course." Requirements engineering is an important development activity within any software development cycle and therefore, understanding and satisfying stakeholder needs and wants is the difference between the success and failure of a product. The authors who have taught a course on requirements engineering many times discuss the methodologies they have used and present the pros and cons of each methodology and reflect on what worked and what did not in teaching requirements to undergraduate engineering students.

Adapting Agile Practices

Zainab Masood, Rashina Hoda and Kelly Blincoe present "Adapting Agile Practices in University Contexts." The authors describe the constraints the students faced while applying agile practices in a university course including difficulty in setting up common time for all team members to work together, limited availability of customer due to busy schedule and the modifications the students introduced to adapt agile practices to suit the university context, such as daily stand-ups with reduced frequency, combining sprint meetings, and rotating scrum master from team. The authors also summarize the effectiveness of these modifications based on reflection of the students and offer recommendations for educators and students. The authors' findings and recommendations will help educators and students better coordinate and apply agile practices on industry-based projects in university contexts.

Collaborative and Teamwork Software Development

Claudia Raibulet and Francesca Arcelli Fontana present "Collaborative and Teamwork Software Development in an Undergraduate Software Engineering Course." Two essential elements of successful software development are collaboration and teamwork. The authors describe their experience in stimulating collaboration and teamwork activities of students in the context of a software engineering course at the third year of an undergraduate program in computer science program. The students were asked to develop a software project in teams of 3–5 students for the final exam of the course. The students were also asked to perform project management tasks (e.g., the Gantt) using the Microsoft Project tool. At the end of the course, the authors gathered the student feedback through a questionnaire on their collaboration and teamwork experience. From their feedback, the students were enthusiastic about working in teams for their project development and about learning how to use tools which are exploited not only in the academic world but also in industry.

Acknowledgment

We thank the organizers of the conference, the authors, the reviewers (CSEE&T and the *JSS*), W.K. Chan who is in charge of the special issues of the *JSS* and the editorial office of the *JSS* for their assistance and for making this special issue possible. We enjoyed working with the above individuals and hope you enjoy their collective contributions.

Hossein Saiedian*
Guest Editor
The University of Kansas, USA

Hironori Washizak
Guest Editor
Waseda University, Japan

*Corresponding author.
E-mail address: saiedian@ku.edu (H. Saiedian)
Accepted 7 September 2018