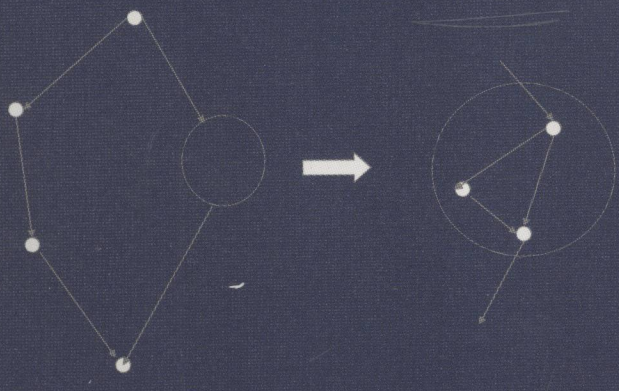
Paola Inverardi
Mehdi Jazayeri (Eds.)

Tutorial

LNCS 4309

Software Engineering Education in the Modern Age

Software Education and Training Sessions
at the International Conference
on Software Engineering, ICSE 2005
St. Louis, MO, USA, May 2005, Revised Lectures



Springer

Paola Inverardi Mehdi Jazayeri (Eds.)

Software Engineering Education in the Modern Age

Software Education and Training Sessions
at the International Conference
on Software Engineering, ICSE 2005
St. Louis, MO, USA, May 15-21, 2005
Revised Lectures

Springer

Volume Editors

Paola Inverardi
University of L'Aquila
Computer Science Department
67010 L'Aquila, Italy
E-mail: inverard@di.univaq.it

Mehdi Jazayeri
University of Lugano
and Technical University of Vienna
E-mail: mehdi.jazayeri@unisi.ch

Library of Congress Control Number: 2006938014

CR Subject Classification (1998): K.3, K.4, D.2, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-68203-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-68203-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11949374 06/3142 5 4 3 2 1 0

Preface

Software Engineering is a multifaceted and expanding topic. It aims to provide theories, methods and tools to tackle the complexity of software systems, from development to maintenance. Its complexity is made even more severe today by rapid advances in technology, the pervasiveness of software in all areas of society, and the globalization of software development. The continuous expansion of the field presents the problem of how to keep up for practitioners. For educators, the key questions are how should software engineers be educated and what are the core topics and key technologies?

Even looking only at the last decade, the tremendous changes that have taken place in the software engineering industry, and in the industrial world in general, raise many questions. What are the effects of: Outsourcing? Distributed software development? Open source? Standardization? Software patents? Model-driven development? How should these developments change the way we teach software engineering? Should textbooks be updated? Should software engineering play a different role in the computer science curriculum, for example, be more pervasive? How are instructors in universities handling these issues?

All these issues were discussed at the Software Education and Training sessions at the International Conference on Software Engineering (ICSE 2005) by leading researchers, educators, and practitioners in software engineering, who presented their—sometimes controversial—views and insights on software engineering education in the new millennium. In this volume we have collected some of the most representative and innovative approaches that were presented at the workshop. The authors revised their papers based on discussions at the conference and the comments they received from the reviews. Together, these papers show the state of the art and practice and the significant challenges facing our field in educating the next generation of software engineers.

The contributions are grouped in two parts. The first part discusses the present. It is introduced by two papers that discuss respectively the limits and the realities of today's software engineering education. The following four papers address the critical problem of teaching software modeling and design, that is, the problem of teaching at the same time, creativity and abstraction, rigorous specifications and easy formalization. The second part of the book deals with the future. In the two introductory papers, representatives from industry and academia, respectively, give their perspectives on the future. The last four papers address different challenges of future software engineering education. On-line education, in-context software engineering education, education to master outsourcing and other distributed software development, organizational software engineering are all concerned with widening the scope of interest of software engineers, intersecting with other disciplines and sciences.

This book provides a snapshot of the state of software engineering education at the beginning of the twenty-first century. It is a good starting point for software engineering researchers and educators alike. It is also a source of ideas for instructors who are looking to improve their software engineering courses.

This book is the result of the work of many people. Paola Inverardi and Mehdi Jazayeri organized the Software Engineering Education and Training Track at ICSE 2005. They requested contributions from the research and education community. The organizers selected the most appropriate contributions and invited the authors to present their positions at the conference. The presentations were followed by active discussions from the audience. In all, a few hundred people participated in the three days of meetings. After the meeting, we asked several of the participants to prepare papers on their contributions for this volume.

We would like to thank all the people who submitted their work to the conference, the people who participated in the sessions, and the authors of the present volume. We also would like to thank Catalin Roman, General Chair of ICSE 2005, who asked us to organize the track. Finally, we would like to thank Jochen Wuttke for helping us to prepare this volume.

September 2006

Paola Inverardi
Mehdi Jazayeri

Lecture Notes in Computer Science 4309

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Lecture Notes in Computer Science

For information about Vols. 1–4242

please contact your bookseller or Springer

Vol. 4345: N. Maglaveras, I. Chouvarda, V. Koutkias, R.W. Brause (Eds.), Biological and Medical Data Analysis. XIII, 496 pages. 2006. (Sublibrary LNBI).

Vol. 4338: P. Kalra, S. Peleg (Eds.), Computer Vision, Graphics and Image Processing. XV, 965 pages. 2006.

Vol. 4337: S. Arun-Kumar, N. Garg (Eds.), FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science. XIII, 430 pages. 2006.

Vol. 4333: U. Reimer, D. Karagiannis (Eds.), Practical Aspects of Knowledge Management. XII, 338 pages. 2006. (Sublibrary LNAI).

Vol. 4331: G. Min, B. Di Martino, L.T. Yang, M. Guo, G. Ruenger (Eds.), Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops. XXXVII, 1141 pages. 2006.

Vol. 4330: M. Guo, L.T. Yang, B. Di Martino, H.P. Zima, J. Dongarra, F. Tang (Eds.), Parallel and Distributed Processing and Applications. XVIII, 953 pages. 2006.

Vol. 4329: R. Barua, T. Lange (Eds.), Progress in Cryptology - INDOCRYPT 2006. X, 454 pages. 2006.

Vol. 4326: S. Göbel, R. Malkewitz, I. Iurgel (Eds.), Technologies for Interactive Digital Storytelling and Entertainment. X, 384 pages. 2006.

Vol. 4325: J. Cao, I. Stojmenovic, X. Jia, S.K. Das (Eds.), Mobile Ad-hoc and Sensor Networks. XIX, 887 pages. 2006.

Vol. 4319: L.-W. Chang, W.-N. Lie, R. Chiang (Eds.), Advances in Image and Video Technology. XXVI, 1347 pages. 2006.

Vol. 4318: H. Lipmaa, M. Yung, D. Lin (Eds.), Information Security and Cryptology. XI, 305 pages. 2006.

Vol. 4313: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods. IX, 197 pages. 2006.

Vol. 4312: S. Sugimoto, J. Hunter, A. Rauber, A. Morishima (Eds.), Digital Libraries: Achievements, Challenges and Opportunities. XVIII, 571 pages. 2006.

Vol. 4311: K. Cho, P. Jacquet (Eds.), Technologies for Advanced Heterogeneous Networks II. XI, 253 pages. 2006.

Vol. 4309: P. Inverardi, M. Jazayeri (Eds.), Software Engineering Education in the Modern Age. VIII, 207 pages. 2006.

Vol. 4307: P. Ning, S. Qing, N. Li (Eds.), Information and Communications Security. XIV, 558 pages. 2006.

Vol. 4306: Y. Avrithis, Y. Kompatsiaris, S. Staab, N.E. O'Connor (Eds.), Semantic Multimedia. XII, 241 pages. 2006.

Vol. 4305: A.A. Shvartsman (Ed.), Principles of Distributed Systems. XIII, 441 pages. 2006.

Vol. 4304: A. Sattar, B.-H. Kang (Eds.), AI 2006: Advances in Artificial Intelligence. XXVII, 1303 pages. 2006. (Sublibrary LNAI).

Vol. 4302: J. Domingo-Ferrer, L. Franconi (Eds.), Privacy in Statistical Databases. XI, 383 pages. 2006.

Vol. 4301: D. Pointcheval, Y. Mu, K. Chen (Eds.), Cryptology and Network Security. XIII, 381 pages. 2006.

Vol. 4300: Y.Q. Shi (Ed.), Transactions on Data Hiding and Multimedia Security I. IX, 139 pages. 2006.

Vol. 4296: M.S. Rhee, B. Lee (Eds.), Information Security and Cryptology – ICISC 2006. XIII, 358 pages. 2006.

Vol. 4295: J.D. Carswell, T. Tezuka (Eds.), Web and Wireless Geographical Information Systems. XI, 269 pages. 2006.

Vol. 4294: A. Dan, W. Lamersdorf (Eds.), Service-Oriented Computing – ICSOC 2006. XIX, 653 pages. 2006.

Vol. 4293: A. Gelbukh, C.A. Reyes-Garcia (Eds.), MICAI 2006: Advances in Artificial Intelligence. XXVIII, 1232 pages. 2006. (Sublibrary LNAI).

Vol. 4292: G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. Nefian, G. Meenakshisundaram, V. Pascucci, J. Zara, J. Molineros, H. Theisel, T. Malzbender (Eds.), Advances in Visual Computing, Part II. XXXII, 906 pages. 2006.

Vol. 4291: G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. Nefian, G. Meenakshisundaram, V. Pascucci, J. Zara, J. Molineros, H. Theisel, T. Malzbender (Eds.), Advances in Visual Computing, Part I. XXXI, 916 pages. 2006.

Vol. 4290: M. van Steen, M. Henning (Eds.), Middleware 2006. XIII, 425 pages. 2006.

Vol. 4289: M. Ackermann, B. Berendt, M. Grobelnik, A. Hotho, D. Mladenič, G. Semeraro, M. Spiliopoulou, G. Stumme, V. Svatek, M. van Someren (Eds.), Semantics, Web and Mining. X, 197 pages. 2006. (Sublibrary LNAI).

Vol. 4288: T. Asano (Ed.), Algorithms and Computation. XX, 766 pages. 2006.

Vol. 4286: P. Spirakis, M. Mavronicolas, S. Kontogiannis (Eds.), Internet and Network Economics. XI, 401 pages. 2006.

Vol. 4285: Y. Matsumoto, R. Sproat, K.-F. Wong, M. Zhang (Eds.), Computer Processing of Oriental Languages. XVII, 544 pages. 2006. (Sublibrary LNAI).

Vol. 4284: X. Lai, K. Chen (Eds.), Advances in Cryptology – ASIACRYPT 2006. XIV, 468 pages. 2006.

Vol. 4283: Y.Q. Shi, B. Jeon (Eds.), Digital Watermarking. XII, 474 pages. 2006.

Vol. 4282: Z. Pan, A. Cheok, M. Haller, R.W.H. Lau, H. Saito, R. Liang (Eds.), Advances in Artificial Reality and Tele-Existence. XXIII, 1347 pages. 2006.

Vol. 4281: K. Barkaoui, A. Cavalcanti, A. Cerone (Eds.), Theoretical Aspects of Computing - ICTAC 2006. XV, 371 pages. 2006.

Vol. 4280: A.K. Datta, M. Gradinariu (Eds.), Stabilization, Safety, and Security of Distributed Systems. XVII, 590 pages. 2006.

Vol. 4279: N. Kobayashi (Ed.), Programming Languages and Systems. XI, 423 pages. 2006.

Vol. 4278: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, Part II. XLV, 1004 pages. 2006.

Vol. 4277: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, Part I. XLV, 1009 pages. 2006.

Vol. 4276: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, Part II. XXXII, 752 pages. 2006.

Vol. 4275: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, Part I. XXXI, 1115 pages. 2006.

Vol. 4274: Q. Huo, B. Ma, E.-S. Chng, H. Li (Eds.), Chinese Spoken Language Processing. XXIV, 805 pages. 2006. (Sublibrary LNAI).

Vol. 4273: I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, L. Aroyo (Eds.), The Semantic Web - ISWC 2006. XXIV, 1001 pages. 2006.

Vol. 4272: P. Havinga, M. Lijding, N. Meratnia, M. Wegdam (Eds.), Smart Sensing and Context. XI, 267 pages. 2006.

Vol. 4271: F.V. Fomin (Ed.), Graph-Theoretic Concepts in Computer Science. XIII, 358 pages. 2006.

Vol. 4270: H. Zha, Z. Pan, H. Thwaites, A.C. Addison, M. Forte (Eds.), Interactive Technologies and Sociotechnical Systems. XVI, 547 pages. 2006.

Vol. 4269: R. State, S. van der Meer, D. O'Sullivan, T. Pfeifer (Eds.), Large Scale Management of Distributed Systems. XIII, 282 pages. 2006.

Vol. 4268: G. Parr, D. Malone, M. Ó Foghlú (Eds.), Autonomic Principles of IP Operations and Management. XIII, 237 pages. 2006.

Vol. 4267: A. Helmy, B. Jennings, L. Murphy, T. Pfeifer (Eds.), Autonomic Management of Mobile Multimedia Services. XIII, 257 pages. 2006.

Vol. 4266: H. Yoshiura, K. Sakurai, K. Rannenberg, Y. Murayama, S. Kawamura (Eds.), Advances in Information and Computer Security. XIII, 438 pages. 2006.

Vol. 4265: L. Todorovski, N. Lavrač, K.P. Jantke (Eds.), Discovery Science. XIV, 384 pages. 2006. (Sublibrary LNAI).

Vol. 4264: J.L. Balcázar, P.M. Long, F. Stephan (Eds.), Algorithmic Learning Theory. XIII, 393 pages. 2006. (Sublibrary LNAI).

Vol. 4263: A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, Y. Saygın (Eds.), Computer and Information Sciences – ISCIS 2006. XXIII, 1084 pages. 2006.

Vol. 4262: K. Havelund, M. Núñez, B. Wolff, G. Roşu (Eds.), Formal Approaches to Software Testing and Runtime Verification. VIII, 255 pages. 2006.

Vol. 4261: Y. Zhuang, S. Yang, Y. Rui, Q. He (Eds.), Advances in Multimedia Information Processing - PCM 2006. XXII, 1040 pages. 2006.

Vol. 4260: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering. XII, 778 pages. 2006.

Vol. 4259: S. Greco, Y. Hata, S. Hirano, M. Inuiguchi, S. Miyamoto, H.S. Nguyen, R. Słowiński (Eds.), Rough Sets and Current Trends in Computing. XXII, 951 pages. 2006. (Sublibrary LNAI).

Vol. 4257: I. Richardson, P. Runeson, R. Messnarz (Eds.), Software Process Improvement. XI, 219 pages. 2006.

Vol. 4256: L. Feng, G. Wang, C. Zeng, R. Huang (Eds.), Web Information Systems – WISE 2006 Workshops. XIV, 320 pages. 2006.

Vol. 4255: K. Aberer, Z. Peng, E.A. Rundensteiner, Y. Zhang, X. Li (Eds.), Web Information Systems – WISE 2006. XIV, 563 pages. 2006.

Vol. 4254: T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, J. Wijsen (Eds.), Current Trends in Database Technology – EDBT 2006. XXXI, 932 pages. 2006.

Vol. 4253: B. Gabrys, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems, Part III. XXXII, 1301 pages. 2006. (Sublibrary LNAI).

Vol. 4252: B. Gabrys, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems, Part II. XXXIII, 1335 pages. 2006. (Sublibrary LNAI).

Vol. 4251: B. Gabrys, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems, Part I. LXVI, 1297 pages. 2006. (Sublibrary LNAI).

Vol. 4250: H.J. van den Herik, S.-C. Hsu, T.-s. Hsu, H.H.L.M. Donkers (Eds.), Advances in Computer Games. XIV, 273 pages. 2006.

Vol. 4249: L. Goubin, M. Matsui (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2006. XII, 462 pages. 2006.

Vol. 4248: S. Staab, V. Svátek (Eds.), Managing Knowledge in a World of Networks. XIV, 400 pages. 2006. (Sublibrary LNAI).

Vol. 4247: T.-D. Wang, X. Li, S.-H. Chen, X. Wang, H. Abbass, H. Iba, G. Chen, X. Yao (Eds.), Simulated Evolution and Learning. XXI, 940 pages. 2006.

Vol. 4246: M. Hermann, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. XIII, 588 pages. 2006. (Sublibrary LNAI).

Vol. 4245: A. Kuba, L.G. Nyúl, K. Palágyi (Eds.), Discrete Geometry for Computer Imagery. XIII, 688 pages. 2006.

Vol. 4244: S. Spaccapietra (Ed.), Journal on Data Semantics VII. XI, 267 pages. 2006.

Vol. 4243: T. Yakhno, E.J. Neuhold (Eds.), Advances in Information Systems. XIII, 420 pages. 2006.

Table of Contents

On Software Engineering Education

State of the Art and Practice: Creativity and Rigor

Challenges for Industries and Academia

Future Directions

Reflections on Software Engineering Education

Hans van Vliet

Vrije Universiteit, Amsterdam
hans@cs.vu.nl

Abstract. The "engineering" focus in software engineering education leaves instructors vulnerable to several traps. It also misleads students as to SE's essential human and social dimensions. In this paper we argue that there's more to SE than engineering. A major challenge is to reconcile the engineering dimension with the human and social dimension.

1 Introduction

In recent years, the SE community has focused on organizing our existing knowledge and finding ways to transform it into a curriculum. These efforts produced SWEBOK (the Guide to the Software Engineering Body of Knowledge; www.swebok.org) and Software Engineering 2004 (http://sites.computer.org/ccse). SWEBOK reflects a widely agreed-upon view of what a software engineer who has a bachelor's degree and four years' experience should know. SE 2004 offers curriculum guidelines for undergraduate SE degree programs. We can view SE 2004 as SWEBOK's education counterpart.

Both SE 2004 and SWEBOK are important milestones resulting from participants' extensive real-world experience and working-group discussions. Both heavily emphasize the 'engineering' in software engineering [1,2,3]. This focus influences the contents of a typical SE course as well as the students' understanding of what SE entails. However, SE has an important social dimension that's easily squeezed out by the omnipresent engineering attitude. Here, I discuss how this limited conception of SE contributes to five assumptions that can trap SE educators:

- An SE course needs an industrial project.
- SE is like other branches of engineering.
- Planning in SE is poorly done relative to other fields.
- The user interface is part of low-level design.
- SWEBOK represents the state of the practice.

The traps idea isn't highly original. Several authors have published similar articles on the myths of formal methods, requirements engineering, and SE programs [4]. In the latter case, the authors discuss whether the new SE degree programs are a silver bullet. The traps I discuss focus on a typical SE course's content and how it represents SE to beginning students. My aim is both to provoke discussion and to highlight the challenges these traps present to SE educators.

P. Inverardi and M. Jazayeri (Eds.): ICSE 2005 Education Track, LNCS 4309, pp. 1–10, 2006.

2 Context

My teaching situation partly determines and bounds the traps I discuss. Typically, Dutch universities don't offer separate computer science (CS) and SE degrees. They have a three-year bachelor's program and a two-year master's program in CS. Most students enroll in the bachelor's program right after high school. The program doesn't have much specialization and usually has one general SE course. Typically, this course includes theory and project work. The master's program generally contains a series of more specialized SE courses.

The Vrije Universiteit rates its SE course's theoretical and practical parts at 4 and 8 ECTS credits, respectively. (In the European Credit Transfer System, 1 ECTS amounts to approximately 28 study hours; a full year is 60 ECTS.) The course lasts 12 weeks. Students are scheduled to take it in the second year of their bachelor's program, which means they have little maturity in CS or SE when they enroll. The course is compulsory for students in CS, AI, and information science. Typically, 150 to 200 students enroll each year.

In terms of SE 2004, we follow a CS-first approach: students aren't introduced to SE in a serious way until the second year. Our course's content strongly resembles that of SE 2004's proposed SE201 course, presenting SE's basic principles and concepts.

3 Software Education Traps

At one time or another, I've fallen into most of the traps discussed here, as have many colleagues with whom I've discussed SE education over the years.

Trap 1: An SE course needs an industrial project
The idea behind this assumption is that we should prepare students for "the real world," which is complex, full of inconsistencies, and ever changing. The real world also involves participants from different domains and has political and cultural aspects. To meet this challenge, we might base projects on real industry examples [5] or introduce obstacles and dirty tricks into student exercises [6]. The question is, how helpful is this?

Student overload. Prior to their second year, students usually have taken courses on programming, data structures, computer organization, and so on. In such courses, instructors typically structure the work clearly and give students unambiguous problems. And too often, the problems have only one right answer. In the SE course, students are suddenly overwhelmed with many new topics. Of course, it might be possible to gently introduce some SE principles in other introductory courses. In practice, this isn't easy in a CS environment.

So, at the start of our SE course, students aren't familiar with requirements engineering (RE) and don't know how to

- write unambiguous requirements or elicit them from stakeholders from other domains,

- prioritize requirements,
- relate requirements to effort (to them, all requirements are equal, regardless of their content), or
- document requirements.

Last but not least, students don't (yet) appreciate RE's value. For example, only a few years ago, I asked students to write a requirements document as their first task. In response, one student complained, "How can I possibly write down what the system does when I haven't programmed the damned thing yet?"

The problem isn't limited to RE. Design, testing, configuration management, quality assurance, and so on all face the same issues. Combining an introduction to all these topics with a real-life case simply asks for too much. Additionally, the students aren't mature enough to appreciate the importance of many SE topics. On one hand, many issues sound obvious: pay attention to documentation, apply configuration control, test thoroughly, and so on. On the other hand, our students have difficulty appreciating issues—such as team organization and cost estimation—that software professionals know from the trenches.

Simplify (when possible). In my SE textbook [7], I use a swimming-lessons analogy. Around 1900, Amsterdam schoolchildren typically learned to swim on the school playground, practicing proper movements while lying on wooden benches. In contrast, my father grew up in the countryside and learned to swim the hard way. His father simply tied a rope around his middle, threw him into the river, and shouted: "Swim." Nowadays, swimming lessons start off gently, in a toddler pool with Mama and plenty of flotation devices. Gradually, the amount of floating material is reduced and the pool gets deeper.

I favor a similar approach. In my SE course, I view my students as toddlers on the SE playground. I concentrate on (at most) a few issues in an orchestrated environment. While I cover all the requisite course topics—and tell my favorite anecdotes—the class project highlights only a few targeted issues. In later years and other courses, students will confront additional real-life aspects. I've often noticed that students' appreciation for my initial SE course comes only years after they've suffered through it.

Design is one key SE issue that instructors can address in an orchestrated way—and that's also a major hurdle for most students. Design is "wicked" [8] because of the following:

- *It has no definite formulation.* We can't neatly separate the design process from the preceding or subsequent phases because they all overlap and influence each other.
- *There is no stopping rule.* No criterion exists to tell us when we've reached the solution.
- *Solutions aren't true or false.* Design involves trade-offs between potentially conflicting concerns. Stakeholders in the design process might define different acceptable solutions rather than one best solution.

The latter point, in particular, opens up interesting project possibilities. An instructor might ask different student groups to design the same system but

with different priorities (with respect to quality requirements or requirements priorities, for example). The groups might later collectively study and discuss the different designs. (My colleague and I have reported on experiences with this approach at the software architecture level [9].)

An interesting and often-applied option is to have a dual program, in which students spend, say, half a year in industry and half a year at school. This reduces the pressure on the university to include "real-life" course elements while also increasing the likelihood that students will appreciate typical SE topics. Unfortunately, that's not an option for all instructors (yours truly included), mainly because of university systems that target full-time students who enter the university right after high school.

Trap 2: SE is like other branches of engineering
All SE texts discuss the similarities between SE and other engineering branches—as well, of course, as the differences (interesting examples compare SE with bridge design [10] and high-pressure steam engines [11]). The overall message, however, is that the similarities prevail.

Engineering's limits. Although the engineering metaphor is useful, there's a downside to it. Our field uses numerous engineering words: building software, requirements, specification, process, maintenance, and so on. Altogether, this induces a model of how we view the software development practice; the engineering metaphor plays an active role in our thought processes [12]. For example, we generally characterize the RE process as follows:

- Information (the requirements) flows from A (the user) to B (the software engineer).
- Good communication doesn't involve any frictions or blockages.
- Good reception of the information involves only extraction.

The underlying model is that requirements exist somewhere; we just have to capture them. Thus, it's a *documentation* issue. If we run into problems, there must be a blockage or breakdown in the communication channel: "Why can't the users express their real demands?"

But there are other options, such as viewing RE as an *understanding* issue. It then becomes a dialog between parties, with the requirements engineer acting as a kind of midwife. The requirements aren't something immutable "out there," but rather, they're constructs of the human mind [13].

Social dimensions. Numerous approaches—such as participatory design, rapid application design, joint application design, facilitated workshops, early user involvement, and so on—try to overcome the traditional, functionalist view's disadvantages with respect to RE. Given the clear assignments students are accustomed to from earlier courses, they often perceive the more open attitude toward RE as confusing. One student spoke for many others in labeling it "a badly organized educational exercise."

At a larger scale, a similar tension exists between the heavyweight, document- and planning-driven life-cycle models from SE's engineering realm and the various

lightweight approaches that emphasize software development's human aspects. Combining the virtues of both is a major challenge. This is true for the state of the practice and even more so for the educational environment, where students are entrenched in the engineering view of the software development world and are not mature enough to perceive the limits of that view.

The latter became apparent recently when several students majoring in multimedia and culture took our SE course. These students clashed with the regular CS students, who held a rather one-sided, traditional view and failed to see and appreciate the nontechnical issues involved.

Today, engineers from all disciplines need social competences, including communication, organization, and conflict-resolution skills. Also, technological possibility is no longer the only driving force behind success. Increasingly, engineers must weigh competing values such as those related to economics, quality of life, and the social and economic impact of job eliminations [14]. We must prepare our students for this future.

Trap 3: Planning in SE is poorly done relative to other fields

Many papers on SE and SE education have quotes like "Approximately 75 percent of all software projects are either late or cancelled." [15] In his wonderful book, *Death March*, Edward Yourdon quotes the Standish Group and gurus such as Capers Jones and Howard Rubin, stating that, "The *average* project is likely to be 6 to 12 months behind schedule and 50 to 100 percent over budget." And "the grim reality is that you should expect that your project will operate under conditions that will almost certainly lead to some degree of death march behavior on the part of the project manager and his or her technical staff." [16] The sometimes explicit, sometimes implicit message is this: A better software education will help, and might eventually even do away with most runaway projects. I question this connection between SE education level and planning accuracy.

Findings on other fields' infrastructure projects. Looking to other fields can be instructive here. Engineers are currently building an expensive high-speed railway connection to carry freight from Rotterdam's harbor to Germany (and beyond). In 1992, officials estimated total costs at 2.3 billion euro; by 2000, they raised the estimate to 4.7 billion euro. Over the same period, the connection's freight estimates continuously dropped. Many people think the connection will never make money.

In 2005, the Dutch parliament launched an inquiry into the project. It first interviewed Danish economist Bent Flyvbjerg and his coauthors Nils Bruzelius and Werner Rothengatter, who studied over 250 international infrastructure projects [17]. They found that nine out of 10 projects underestimate costs, and almost all projects overestimate revenues. The combination makes projects look good and helps ensure decision makers' approval. Now, people naturally overestimate the good and underestimate the bad, particularly in cases of uncertainty. If you ask people whether they think more people die of cancer or diabetes, they'll most likely say cancer. In fact, it's diabetes, which most people consider to be the less dangerous disease. But there are other explanations as well.

Flyvbjerg, Bruzelius, and Rothengatter cite several well-known projects with spectacular overruns:

- Suez Canal (1869): 1,900 percent over budget
- Sydney Opera House (1973): 1,400 percent over budget
- Concorde (first flight in 1969): 1,100 percent over budget
- Panama Canal (1913): 200 percent over budget
- Brooklyn Bridge (1883): 100 percent over budget

On railway projects, they found that the average project overrun is 45 percent. Next come bridges and tunnels, which have an average overrun of 34 percent.

The authors dismiss technical explanations for such project overruns. If it were simply a matter of technology, then statistically, they would have also found projects with cost underruns. They didn't. Likewise, they dismiss psychological explanations related to estimators' natural optimism. If that were true, we could assume that estimators don't learn from past mistakes. The conclusion? Estimators intentionally underestimate project costs for political reasons: the pressure is high, the parties involved have already made a deal, the project "must be done," and so on.

Software analogue. Many of the arguments that hold for infrastructure project cost and schedule overruns are also valid for software development projects. Educating future software engineers to better count function points, engineer requirements, and approach other key tasks won't on its own resolve overrun issues. As Tom DeMarco put it in 1982, "One chief villain is the policy that estimates shall be used to create incentives." [18] This is as true today as it was then.

In one interesting software cost-estimation experiment [19], the authors studied the "winner's curse," which has the following characteristics:

- Software providers differ in their estimate optimism: some are overly optimistic, some are realistic, and some are pessimistic.
- Software providers with overly optimistic estimates tend to have the lowest bids.
- Software clients require a fixed-price contract.
- Software clients tend to select a provider with a low bid.

The resulting contract often delivers low or negative profits to the bidder; it can also be risky for the client. In one experiment [19], for example, Magne Jørgensen and Stein Gromstad asked 35 companies for bids on a certain requirements specification. They then asked four companies to implement the system. They found that the companies with the lowest bids incurred the greatest risks.

Both Flyvbjerg and Jørgensen emphasize the need for careful risk management. As one experienced project manager told me, "Risk management is project management for adults." Risk management definitely deserves a front seat in a full-fledged SE curriculum.

Trap 4: The user interface is part of low-level design

> *We can't worry about these user interface issues now. We haven't even gotten this thing to work yet!—R. Mulligan et al. [20]*

A system's user interface is important: In an interactive system, about half the code is devoted to the user interface. In a recent study, researchers found that 60 percent of software defects arose from usability errors, while only 15 percent related to functionality [21]. In addition, adequate attention to user interface quality can increase sales of e-commerce sites by 100 percent [22]. For Web-based systems, usability goals are business goals. To improve the state of the practice, we should integrate appropriate user interface design techniques into our software development process. The place to start this practice is SE education.

Ignoring human factors. Is the SE community integrating user interface design techniques into the development process? SWEBOK and SE 2004 offer the most relevant answers here. SWEBOK lists human-computer interface (HCI) as a "related discipline" of SE, concerned with understanding the interactions among humans and other system elements. SE 2004 takes a similar position, describing an HCI course in which user interface design concerns topics such as "use of modes" and "response time and feedback."

Both organizations reflect Mulligan and colleagues' limited view of the user interface. This view totally ignores the fact that many current and future software development projects will aim to develop systems in which human use and related human factors are decisive elements of product quality.

A broader view. Interface design and functionality design go hand in hand. We might even say that the user interface *is* the system. There are two main reasons to take this broader view of the user interface. First, the system—and hence its interface—should help users perform certain tasks. The user interface should therefore reflect the task domain's structure. The design of tasks and their corresponding user interfaces influence each other and should be part of the same iterative refinement process. Like quality, the user interface isn't a supplement. Second, dialog and representation alone don't provide users with sufficient information. To work with a system, users sometimes need to know "what's going on behind the screen."

Various studies corroborate the need to better attend to HCI in SE and CS curricula. Timothy Lethbridge [23], for example, addresses the question of what software professionals need to know. He found that HCI is one of the topics with the widest educational knowledge gap. As Lethbridge reports, practitioners called HCI an important topic but one they'd learned little about in school. Nigel Bevan [24] shows that we must expand the traditional quality assurance approach—which emphasizes software's static and dynamic properties—to incorporate quality-in-use aspects that address broader ergonomic issues.

Proponents of traditional SE see user interface design as a separate activity and don't include it in the mainstream software development process model. We need a more eclectic approach in which we attend to user interface issues