Using Software Engineering Concepts and Techniques to Leverage Learning: A Novel Approach

Daniel Berleant[1], Zhong Gu[1], Steve Russell[1], James F. Peters[2], Sheela Ramanna[3], and Hal Berghel[4]
[1]Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa 50011 /
[2]Department of Electrical and Computer Engineering, University of Manitoba, Winnipeg, Manitoba R3T 5V6, Canada /
[3]Department of Business Computing, University of Winnipeg
Winnipeg, Manitoba R3B 2E9, Canada /
[4]Department of Computer Science, University of Nevada, Las Vegas, Nevada 89154

This paper describes an approach to integrating software engineering concepts and principles into the Electrical and Computer Engineering (ECE) and Computer Science (CS) curricula. Our philosophy is to apply software engineering techniques throughout the ECE/CS curricula to leverage learning in non-software engineering courses. Our technique is to seek out faculty interested in innovative teaching techniques, consult with them to identify some way that they and we feel a course they are teaching could benefit pedagogically from some application of software engineering, and work with them to make that happen. The chief intended result is to leverage learning in diverse courses, thereby benefiting pedagogy of non-software engineering topics. An auxiliary important result is to increase awareness among both students and faculty of the software engineering body of knowledge.

Many software engineering approaches to understanding and solving problems in the software life cycle can also address a variety of learning needs across disciplines in ECE and CS. For example, there are software engineering techniques that can emphasize visualization (benefiting students who respond to the visual modality), logical sequences (benefiting sequential learners), summarizations (benefiting global learners), and others. Additionally, general issues of teamwork and the engineering life cycle can be addressed.

We have applied our approach to a diverse set of electrical engineering and computing courses at four universities in the US and Canada, and based on those experiences we believe we have identified a win-win paradigm that can be a model for integration of software engineering concepts into electrical engineering and computing curricula.

Introduction

Software engineering has rapidly become a major topic in computing education. Departments of software engineering and degree programs in software engineering are increasing in number, and guidelines for software engineering education are receiving increasing attention (Barnes 1998[3]; Bagert et al. 1999[2]; Joint Task Force on Computing Curricula 2000[10]). As part of this

trend, programs in computing and engineering disciplines such as electrical engineering have been increasing their students' foundation of knowledge either by direct or implicit incorporation of materials from the software engineering life cycle paradigm. This paradigm is characterized by a feedback loop containing requirements specification-design-measure-test-maintain steps (Peters and Pedrycz 2000[18]).

Deliberate application of this paradigm can sometimes lead to growing pains in programs in which this occurs. When software engineering education is viewed as a necessary part of a curriculum, but different and distinct from other curriculum areas, it can end up in unhealthy competition for scarce resources. In contrast, we suggest a much more optimistic vision in which software engineering is made into a positive force for other educational goals and becomes a tool for helping to teach other topics more efficiently. As a valuable side effect this will also help familiarize students and faculty with software engineering concepts.

We are taking software engineering education beyond something that existing programs, often stretched thin already, must add to their list of responsibilities. In particular, we are making software engineering *contribute directly to education in other, non-software engineering contexts* (Peters and Pedrycz 1999[17]; Pedrycz and Peters 1998)[15]. To do this, we draw upon software engineering paradigms and frameworks to aid in teaching students a variety of subject areas that form the traditional bread-and-butter of electrical engineering and computing curricula. In particular, the application of software engineering benefits pedagogy throughout a typical curriculum by better integrating software engineering concepts into the curriculum. As a welcome side effect, this pedagogy will also help to familiarize both students and faculty with software engineering itself. The result of this novel approach to integrating software engineering concepts into a curriculum is the infusion of new approaches to realizing educational goals in both software engineering and non-software engineering areas throughout the curriculum.

State of the field

The "across the curriculum" paradigm is well recognized. One of the best-known examples is that of writing across the curriculum, which has been influential in higher education for a number of years. A number of efforts have specifically addressed computing curricula. Arnow et al.[1] describe teaching distributed computing across the computing curriculum. An NSF-funded effort toward development of teaching social impact and ethics across the computing curriculum spans a number of years and institutions (Martin et al. 1996; Huff et al. 1995; Braxton and Stone).[5,9,12] Closer to our present concern of software engineering, Thompson and Hill (1995) describe teaching functional programming across the curriculum.[23] More recently, a conference was devoted to teaching object orientation across the computing curriculum.[14] Grodzinsky et al. (1998) describe using project teams across the computing curriculum.[7] Cushing (1997) describes cleanroom software engineering techniques across the curriculum.[6] Software engineering as a field has considerably greater breadth than what these efforts are concerned with (Liu and Peters 1999[11]; Peters and Ramanna 1998[19]; Peters et al. 1998[20]; Peters et al. 1998[16]).

Software engineering as a broad field to be taught across the computing curriculum is described

by Placide (1999), Werth (1998), Horan et al. (1997), and McCauley et al. (1995).[8,13,21,25] Such works have emphasized the opportunities presented in computing curricula to teach software engineering. In contrast, our approach focuses on *using* the software engineering body of knowledge to teach *other* topics in computing – a process that uses software engineering to support other educational goals yet facilitates software engineering education as well by providing opportunities for students and faculty to embed software engineering concepts into their ways of thinking about many kinds of problems.

A large proportion of university graduates of electrical engineering and computing (and other) programs ultimately take employment involving software development, yet there is a continuing shortage of competent software developers (Strigel 1999).[22] At the same time, software quality is receiving increasing attention as software systems occupy increasingly critical positions in the societal infrastructure (Voas 1999).[24] Competence in software engineering may well be on its way to becoming as essential to the electrical engineering and computing program graduate as competence in writing is to all graduates. Thus, strategies of incorporating software engineering across the curriculum continue to form a timely area of investigation. Our proposed strategy is novel in its focus on applying software engineering principles to help teach diverse areas including such areas as signal analysis, circuit design and telecommunication system design. As a consequence, software engineering can benefit other educational goals, while software education itself benefits because people would be gaining experience in using software engineering principles in varied contexts. We believe the time is ripe for this novel pedagogical strategy both for its own benefits as well as for its potential to serve as a model for analogous efforts in other fields.

Applying the Approach

The feasibility of the approach has been established through an ongoing project in which we apply software engineering to assist in the pedagogy of diverse courses at Iowa State University, University of Manitoba, University of Winnipeg, and University of Nevada, Las Vegas. A review of courses and topics affected is given next, followed by a procedure we have developed for applying the approach to a particular course.

Table 1 lists courses that we have addressed, and Table 2 lists courses that are soon to be addressed and for which plans for how to do this are fully in place. These tables show the diversity of courses to which our approach can be applied. In fact, we hope eventually to address the full range of courses in electrical engineering and computing curricula (except for software engineering courses themselves which are automatically addressed by definition).

How the courses were addressed

This section describes the relevant aspects of each of the affected courses and how software engineering concepts were applied to those aspects. These descriptions could be used as a guide to applying the approach to courses at other institutions.

Curriculum, University	Course	# of Students	Instructor	Subject
CE, ISU	Cpr E 489	127	S. Russell	Computer networking
EE, ISU	EE 424	43	J. Dickerson	Digital signal processing
CE, ISU	Cpr E 305	78	A. Somani	Computer system organization and design
CE, ISU	Cpr E 308	76	J. Davis	Operating systems
CE, ISU	Cpr E/EE 465	37	W. Black	VLSI layout and design
CE, UM	24.374	84	J.F. Peters	System engineering principles
CE, UM	24.446	40	J.F. Peters	Parallel processing
EE, UM	24.771	30	J.F. Peters	Optimal control
Applied Computing, UW	91.4901	60 (2 terms)	S. Ramanna	Senior project course
CS, UNLV	470/670	10	H. Berghel	Multimedia systems design
CS, UNLV	341&341L	30	H. Berghel	Internet programming

Table 1. Courses already addressed.

Curriculum, University	Course	# of Students	Instructor(s)	Subject
CE, ISU	Cpr E 184x	72	D. Jacobson	Computer engineering & problem solving
CE &EE, ISU	Cpr E/EE 491 & 91.4901	20	D. Berleant, J. Lamont, and S. Russell	Senior design and project
CE, UM	24.375 (UM)	80	S. Silverman	System engineering principles II
CE & EE, UM	24.765 (UM)	30	J.F. Peters	Intelligent systems design

Table 2. Courses being addressed.

Digital signal processing (Senior level) The lab exercises often rely on fragments of code developed in previous labs using tools such as Matlab. To facilitate this process, students were given written instructions at the beginning of the semester on segmenting and commenting their code to ease its reuse in future labs, and required to comply with them as they developed code.

Computer engineering & problem solving (Freshman level) The syllabus has been updated to incorporate three software engineering-based changes, which will be taught in spring 2001. The change is based on teaching teamwork and the importance of good design, and involves two 2-person teams with each team developing a design for software that controls a robot's path of motion, which is passed on to another team to grade, then implement (using a real robot), and finally regrade. This pair of grades, generated by other students in the class, is incorporated into

the grade for the exercise that is ultimately recorded. The implementation step as well as the two grading steps are intended to provide the opportunity for students to reflect on what makes a good design.

The second syllabus modification is based on teaching the need for team organization, to handle the rapid increase in the number of communication paths as team size increases. The class will be divided into several teams, each with a different organization. After each team performs the task of summing a rather long list of small integers, the class will discuss their varied experiences with problem decomposition, team organization and communication.

The third syllabus modification is based on teaching the general structure of the software engineering life cycle and the effort reduction benefits created by dealing with problems as early in the life cycle as possible. Small teams will be given specifications for the volume of containers they are to build out of paper and scotch tape during the class session, following which they will be asked to generate a design and then implement it in class. Different teams will be subjected to modest specifications changes at various points in their task (including after implementation is completed, to incorporate the maintenance phase of the life cycle). This will be followed by a class discussion focusing on the relationship between how late in the life cycle the specifications changes are made and the effort required to comply with the change.

Operating systems (Junior level) Students did a model code review exercise. An introductory lecture communicated evidence to them that such non-execution based testing methods have been shown to improve the efficiency of software development. The intent was to give them the motivation to write software for lab exercises ahead of time and the knowledge to be able to collaborate in reviewing it in lab before trying to compile and run it.

Computer system organization and design (Junior level) This course makes extensive use of hardware simulation models which are developed and run by the students. This lends a software development character to the course for which software engineering principles can be beneficial. An early homework exercise involving model development was replaced with a new one in which students were given a model incorporating relevant and educational faults ("bugs"), such that identifying and fixing them would help students in debugging their own models later. In addition, a lecture and in-class exercise was given on the advantages of good specifications and design prior to implementation. This discussion emphasized the savings in overall effort that are achievable by dealing with problems early in the development life cycle when they are easier to fix, rather than later when fixing them tends to require a much greater expenditure of effort.

VLSI layout and design (Senior level) Students need to be able to deal with bugs and other user-unfriendly characteristics of the modeling software they use in labs. This often leads to problems in finishing the labs. To address this problem, the students were explicitly required to report such problems – and their solutions – in their lab notebooks. Doing this appears to have the following positive effects: (1) it makes the students think about the problem, its cause and its solution, thereby giving them the ability to handle similar problems more efficiently in the future, (2) it impresses on students that thinking about such issues can be useful, and (3) the

instructor and TAs are made aware of the frequently occurring problems and how to deal with them sooner, which helps them to help the students to deal with them. We concurrently also developed a Web-based problem reporting system for use in future semesters (at http://wireless.ee.iastate.edu/CE465/).

Senior design (Senior level) Because the course is organized around teams of students doing projects from the beginning of the life cycle through the implementation phase, good team organization is valuable. Also, because the entire life cycle is exercised during the course, understanding the importance of taking care of problems as early in the life cycle as possible is valuable. To address these factors, we meet with our senior design groups and explain the number of communication paths that exist in their team. With this as motivation, we provide a team organization to control that number of communication paths and show what the new number is. To address the life cycle factor, we keep the team focused on following the phases of the life cycle well by having them report at each meeting where they are in the life cycle (which phase and where in the phase), and insist that they revise previous reports when appropriate. Thus, when they are working on the design and discover that a specification needs revision, they must revise the specifications document. This is something that would be easy for them (and us) to let slide if we were not intent on following a good life cycle model.

Internet programming (Junior level) Students were assigned individual programming assignments organized by theme so that each weekly programming assignment was built upon the previous week's effort. The final month of term, the programming assignment model was completely changed to a team approach, where each team of 3 or 4 students was given a multi-module programming assignment with considerable design constraints and encouraged to find their own optimal way of dividing the level of effort. This approach has the dual advantage of both encouraging and verifying individual capabilities while at the same adding a real-world, team-oriented dimension to the software development effort. In order to monitor every member's contribution to the team project, final grade is withheld until every team member assesses the level and nature of contribution of the other members of their team via confidential communication with instructor.

Multimedia systems design (Senior and Graduate level) In this case, students from computer science and content-oriented disciplines such as film, art and music, are required to work as teams on a semester project, with each team demonstrating mastery of the individual assignment components (e.g., image manipulation and animation; digital audio recording, rendering and modeling; mixing and editing; analog and digital video capture and editing; and scripting, scoring and multimedia integration), while concurrently taking on a leadership/tutorial role in one particular aspect of the project. In this way, the students have the combined experience of developing a cohesive team project and also communicating technical expertise to those less skilled. This parallels the actual software development environment in which every participant is both practitioner and educator.

System engineering principles (Junior Level) This course focuses on an object-oriented approach to system engineering based on paradigms found in software engineering (capability maturity model, planning, configuration management, cost estimation, process model, feedback

control, visualization, vanilla frameworks, problem analysis, description, identification of system architectures, detailed design, verification, validation, measurement, testing, reliability, computer-human interfaces, reengineering, maintainability). It provides an essentially normative ("how to") approach to developing systems. It relies on the use of metrics to measure features of proposed and prototypes of software products, and to gain an understanding of how one might design maintainable software. These metrics include effort, complexity, cost, risk, reusability, design-length, and maintainability. The design process is examined in the context of software configuration management, system description with statecharts, architectural description with the Communicating Sequential Processes (CSP) language, reverse engineering (extracting descriptions from code), and forward engineering (adding features to an existing system and starting from scratch). Design begins with consideration of the architecture (structure) of system, and moves toward the creation of a working prototype. Rapid prototyping and incremental design of software are emphasized. Java is used in designing user interfaces. This course also includes consideration of the design and re-design of software metrics such as design length and program effort. This form of design is reflected in one of the labs and in two of the three exams. The focus in designing new software metrics is on adapting a classical model to the needs and features of object-oriented, software portion of system designs containing classes and methods. In addition, the labs for this course include a reverse engineering problem and a number of simple forward engineering problems. A web page for this course include assignments, laboratories, lectures, and exams (see http://www.ee.umanitoba.ca/programs/undergrad/c24374/index.html).

Parallel processing (Senior level) This course focuses on the design of parallel processing systems. Topics include Flynn taxonomy, parallel architectures, parallel processing paradigms, design process, implementation, speedup, performance metrics, computation models, tasks, data, communication. The course includes the design and implementation of a system of hunter and prey robots, which interact with each other and their environment. The implementation will be a massively parallel system where animats learn to cooperate, find or avoid each other. The user interface for this system will use Java.

Optimal control (Graduate level). This course focuses on the design, measurement and optimization of a variety of adaptive linear controllers. Methods and paradigms (planning, cost estimation, requirements, architectures, measures, testing) from software engineering are used to organize, prototype and measure controller designs. Methods and theory from fuzzy sets, rough sets, evolutionary computing, and neural computing are used in various hybrid linear controllers. The project for this course was to consider approaches to the design adaptive attitude (yaw, pitch and roll) controllers for a small geocentric satellite. An adaptive pitch controller is to be designed using a combination of tuner design (gain selecting) based on rough set theory and a classical PD controller.

Many alternative treatments are feasible

The preceding listing describes what was done in a number of courses, but it is important to note that in most cases there are numerous alternative ways that a course could be addressed. Often there are even various alternative ways that a single topic in a course could be addressed. To illustrate, one additional course is provided next. For this course, the various possibilities

that were identified as feasible for *just a single topic* in the course are described (the possibility that was ultimately used is also noted).

Computer networking (Senior level) Consider the problem of learning to understand a digital communication protocol. Various software engineering concepts could potentially be used as pedagogical tools to aid this understanding. A number of them are reviewed now.

The software life cycle begins with a rough account of what the product will do (often called the **requirements analysis or functional requirements**). Applied to the problem of teaching students a new protocol, this suggests beginning by reviewing the point of the protocol, in essence providing them with a requirements analysis for the protocol. **Scenarios** and **rapid prototyping** might be good choices in illustrating the requirements. A rapid prototype approach might illustrate the protocol in slow motion, for example. However it should be mentioned in this regard that current rapid prototyping environments are not without shortcomings and should be used with a considerable measure of caution, particularly within an educational setting.[4]

The next thing in a typical software engineering life cycle would be the detailed requirements, or **specifications**. Specifications tell what a piece of software will do but not how it will do it. Thus specifications in this case would flesh out the point of the protocol with all significant details. Once the students understand what the protocol does via the specs, they should be in a better position to understand how it does what it does. Depending on the problem, appropriate and widely used software engineering techniques useful in communicating the specifications include **data flow diagrams, entity-relationship modeling, finite state machines**, and **Petri nets**. For the communications protocol problem, finite state machines might be a good choice due to their ability to describe the problem while being understandable to the students.

Describing how the protocol does what it does corresponds to the **design** phase of the software life cycle. The first subphase of design is **architectural design** in which the software system is broken down into **modules**. From the standpoint of a digital communication protocol, the modules might include one that runs on the sender and one on the receiver. Students would be better equipped to understand the details of how the communication protocol works if they understand a modular breakdown of it first. A software engineering approach to doing this would likely use either **data flow diagrams**, or Unified Modeling Language (UML) **sequence diagrams** as a description language for giving an understandable picture of a communication protocol. The sequence diagram concept was the one we chose and used in this course. Other problems might call for **transaction analysis, abstract data types**, **class diagrams** with method descriptions, **collaboration graphs,** or some combination.

The second design subphase is the detailed design, in which the details are given but not the actual program code. A **stepwise refinement** approach to presenting the detailed design might be helpful. This amounts to presenting something in successively more detail. This could be done with **flow charts** or in **Program Description Language** (**PDL**), often called **pseudocode**. PDL is basically a description that uses control statements of a given programming language, with other aspects of the code described in concise English rather than actual code. Flow charts

might be good for visual students, while PDL might be more suitable for sequential learners or ones who learn well from printed text. Perhaps presenting a flow chart or PDL, and then converting it to the other form during a lecture, would be useful.

The reader may notice that while many of the system development methods are specific to software engineering, some description methodologies (such as state machines and various forms of diagrams) as well as design methodologies (including prototyping, incremental development, measurement, and testing) are more generic to engineering. This is good, since this non-empty intersection of software engineering and traditional engineering methods will facilitate their understanding and use across a typical university curriculum. The reader may also notice that it is not necessary to discuss details of the protocol itself as we describe how software engineering concepts could be brought to bear in teaching the protocol - it is enough just to know that it is a communication protocol that is under consideration. This seems to be true for many topics, and significantly facilitates the process of disseminating the strategy of software engineering throughout the curriculum. From these various alternatives, sequence diagrams were chosen for use in the course. Future evolution of the course could incorporate other alternatives as well.

Procedure for leveraging learning with software engineering in a given course

Those wishing to get more ideas for their own courses can find many more details of the Iowa State portion of the project at http://class.ee.iastate.edu/berleant/home/research/SE/index.htm.

Applying software engineering in diverse courses requires close coordination with the faculty who teach those courses. We have addressed this by developing a procedure incorporating the required degree of coordination. While systematic evaluation of this procedure is needed, as it currently stands it has worked and appears to constitute a strong foundation for any future refinements. It is given next as a six step process.

Step 1: *Recruit interested faculty.*
Step 2: *In consultation with the instructor, identify promising course topics.*
Step 3: *Generate topic treatment alternatives.*
Start feedback loop
    Step 4 (includes loop test): *Consult with the faculty to narrow the set of alternatives under consideration.*
    Step 5: *Develop alternatives still under consideration further.*
End feedback loop

Step 6: *Use the results to leverage learning.*

Evaluation

There are two main aspects to evaluation of this work. One is to measure how satisfactory the procedure is that applies software engineering to individual courses, and the other is to measure the benefits in individual courses.

To evaluate the six-step procedure, each of the steps should be evaluated separately. Since all of

the steps must succeed well for the procedure as a whole to succeed well, evaluating each step will help identify any steps that constitute bottlenecks and hence should be modified to work better. Since all of the steps must work for the procedure to work, all of the steps will ideally work well. Quantitative measures of the success of each step have been planned based on such raw data as the number of promising course topics identified, the number of alternative software engineering-based treatments of a topic that are found, etc. Because this aspect of the research has yet to be carried out, we defer details to future accounts of the work.

To evaluate the success of our approach on individual courses, a somewhat ad hoc approach is necessary; evaluations can however be done in a well-disciplined manner. Because of the panoply of ways the software engineering field can address different pedagogical issues in diverse courses, there is no single assessment method that would apply to all cases. An evaluation plan is needed for each affected course. Each evaluation plan will be developed in collaboration with the instructor of the course, in accordance with the following outline.

Start loop

- *A) Schedule a consultation.*
- *B) Consultation.*
- *C) Recording to a repository of evaluation activities and measurements.*

End loop if satisfactory evaluation plan is in place for the course.
D) Perform the evaluation.

Conclusion

The work we have done up to this point has shown the feasibility of the approach, but more remains to be done. One next step is to evaluate the results on student learning. Another next step is to facilitate use of the approach in other courses and universities. Eventually we hope to have workshops to help others adapt the approach to their own situations. Ultimately, we hope that the approach of integrating software engineering concepts into the curriculum as a tool to leverage learning in non-software engineering courses will become widely visible and used in curricula at many universities. We believe that the general paradigm of "across the curriculum" educational approaches could benefit significantly from a key component of our approach: teaching and learning of a target discipline (in the present case software engineering) by using it, in part genuinely altruistically, to support the pedagogical goals of other disciplines.

Bibliography

1. Arnow, D.M., P. Whitlock, and C. Tretkoff, Using Modules to Integrate Distributed Computing in the Undergraduate CS Curriculum, First Annual Northeastern Small College Computing Conference (CCSCNE 96), April 19-20, 1996, University of Hartford, CT. Consortium for Computing in Small Colleges. http://www.sci.brooklyn.cuny.edu/~arnow/ED/CCSCNE96/NSCCC96.PAPER.html.

2. Bagert, D.J., T.B. Hilburn, G. Hislop, M. Lutz, M. McCracken, and S. Mengel, Guidelines for Software Engineering Education Version 1.0, Technical Report CMU/SEI-99-TR-032, Software Engineering Institute, Carnegie-Mellon University, 1999. http://www.sei.cmu.edu/publications/documents/99.reports/99tr032/99tr032abstract.html

3. Barnes, B., G. Engel, M. Griss, R. LeBlanc, T. Wasserman, and L. Werth, Draft Software Engineering

Accreditation Criteria, Computer 31 (4) (May 1998) 73-77.
4. Berghel, H., "New Wave Prototyping: The Use and Abuse of Vacuous Prototypes," Interactions, 1:2 (1994), pp. 49-54. http://www.acm.org/~hlb/publications/new-wave/new-wave.html
5. Braxton, S. and D. Stone, ImpactCS, http://www.seas.gwu.edu/~impactcs/index.html, Web site.
6. Cushing, J., Seamless Integration of (Cleanroom) Software Engineering Techniques into a Computer Science Curriculum, Integrating Recent Research Results: An NSF-CISE Educational Infrastructure Workshop, July 7-11, 1997, Evergreen State College, Olympia, WA. http://www.evergreen.edu/user/CISE/cleanroom.html.
7. Grodzinsky, F. S., Project Teams: How to Build, Use, and Evaluate Them in Courses Across the Computer Science Curriculum, The Consortium for Computing in Small Colleges Third Annual Northeastern Conference (CCSCNE 98), April 24-25, 1998, Sacred Heart University, Fairfield, CT. Consortium for Computing in Small Colleges. http://orion.ramapo.edu/~csconf/ccscne/98.
8. Horan, P., A. Goold, and Y. Yang, Integrating Software Engineering Throughout the Computing Curriculum, technical report TRC97/01, 1997, Deakin University, School of Computing and Mathematics, Victoria, Australia.
9. Huff, C. R., Martin, C.D. and Project ImpactCS Steering Committee. Computing consequences: A framework for teaching ethical computing (First Report of the Impact CS Steering Committee). *Communications of the ACM*, Vol.38, no.12, p.75-84, Dec., 1995.
10. Joint Task Force on Computing Curricula, Computing Curricula 2001, sponsored by IEEE Computer Society and by ACM, 2000. http://computer.org/education/cc2001/report/index.html.
11. Liu, P. and J.F. Peters, Risk Analysis in Ice-Melting HVDC Transmission Lines, Proc. CCECE'99, Edmonton, Alberta, May, 1999.
12. Martin, C. D., Huff, C. Gotterbarn, D., Miller, K. and Project ImpactCS Steering Committee, Implementing a tenth strand in the computer science curriculum (Second Report of the Impact CS Steering Committee). *Communications of the ACM*, Vol.39, no.12, p. 75-84, Dec., 1996.
13. McCauley, R. A., C. Archer, N. Dale, R. Mili, J. Roberge, and H. Taylor, The Integration of Software Engineering Principles Throughout the Undergraduate Computer Science Curriculum, *SIGCSE Bulletin* 27(1) (1995) 364-365.
14. Ourusoff, N., Organizer, 1st Maine Higher Education Faculty/Student Development Workshop on "Teaching the O-O Paradigm Across the Computing Curriculum," May 1999. http://users.uma.maine.edu/faculty/nourusoff.
15. Pedrycz, W. and J.F. Peters (Eds.), *Computational Intelligence and Software Engineering*. Singapore: World Scientific Publishing Co. Pte. Ltd., 1998.
16. Peters, J.F., R. Agatep, S. Cormier, N. Dack, F. Kaikhosrawkani, N. Lao, O. Orenstein, P. Thang, V. Wan, and W.Y. Wong, Air Traffic Control Trainer Software Development: Concepts, blackboard architecture and Java prototype. Proc. of the Canadian Conf. on Electrical & Computer Engineering (CCECE'98), Waterloo, Ontario, 25-28 May 1998, 601-604.
17. Peters, J.F. and W. Pedrycz, Computational Intelligence. In: J.G. Webster (Ed.), *Encyclopedia of Electrical and Electronic Engineering*. 22 vols. NY: John Wiley & Sons, Inc., 1999, http://www.wiley.com/ee/engineering.htm.
18. Peters, J.F. and W. Pedrycz, *Software Engineering: An Engineering Approach*. New York: John Wiley & Sons, Inc., 2000.
19. Peters, J.F. and S. Ramanna, Time-Constrained Software Cost Control System: Concepts and Roughly Fuzzy Petri Net Model. In: L.A. Zadeh, K. Hirota, E. Sanchez, P.-Z. Wang, and R.R. Yager (Eds.), Advances in Fuzzy Systems—Applications and Theory, Vol. 16. Singapore: World Scientific Publishing Co. Pte. Ltd., 1998, 339-369.
20. Peters, J.F., J. Wong, and S. Ramanna, Evolution of Competing Situated Robots: Concepts, and Experiments with a Java Applet. Proc. SMC'98, San Diego, California, October 1998, 3371-3374.
21. Placide, E., The Merging/Integration of Software Engineering Across the Computer Science Curriculum, ADMI 99 (The Symposium on Computing at Minority Institutions), June 3-6, Duluth, MN. http://cs.fdl.cc.mn.us/admi99/1999SCHEDULE121.html.
22. Strigel, W., What's the Problem: Labor Shortage or Industry Practices?, Guest Editor's Introduction, *IEEE Software* 16 (3) (May/June 1999) 52-54.
23. Thompson, S. and S. Hill, Functional Programming Through the Curriculum, in P. H. Hartel, and R. Plasmeijer, eds., *Functional Programming Languages in Education*, Lecture Notes in Computer Science

number 1022, Springer-Verlag, 1995, 85-102. http://www.cs.ukc.ac.uk/pubs/1995/214/index.html.
24. Voas, J., Certification: Reducing the Hidden Costs of Poor Quality, Guest Editor's Introduction, *IEEE Software* 16 (4) (July/August 1999) 22-25.
25. Werth, L.H., Integrating Software Engineering into Introductory Computer Science Courses, *Computer Science Education* 8 (1) (March 1998).

DANIEL BERLEANT
Daniel Berleant is an Associate Professor in the Electrical and Computer Engineering Dept. at Iowa State University, where he has been since 1999. He received the PhD from University of Texas at Austin in 1991. His interests include software engineering, text mining and interaction, arithmetic on random variable operands, and technology foresight.

ZHONG GU
Zhong Gu received the BS degree and MS degrees in Electrical Engineering in 1995 and 1998, respectively, from Xidian University. He is now a master's degree candidate in Computer Engineering at Iowa State University. His research interests include multimedia browsing systems and use of software engineering techniques to support pedagogy across the EE and CSE curricula. He is a member of the IEEE Computer Society and the IEEE Communication Society.

STEVE RUSSELL
Steve F. Russell is an Associate Professor of Electrical and Computer Engineering at Iowa State University. He is a registered professional engineer and is active in research with industry in the areas of communication systems and signal processing. He spent 13 years in industry prior to joining ISU. Dr. Russell received the B.S. degree in EE from Montana State in 1966 and a Ph.D. from Iowa State in 1978.

JAMES F. PETERS
James F. Peters is in the Department of Electrical and Computer Engineering, University of Manitoba, in Winnipeg, Canada. He received his Ph.D. in 1991. He is an IEEE Millenium award recipient. He has published over 100 papers in software engineering since 1991 and has authored a recent textbook in the subject.

SHEELA RAMANNA
Sheela Ramanna is an Associate Professor and Head of the Department of Business Computing, University of Winnipeg, Manitoba, Canada. She received her Ph.D. in 1991. She has published over 50 papers in software engineering.

HAL BERGHEL
Hal Berghel, PhD, is Professor and Chair of Computer Science at the University of Nevada, Las Vegas. His interests are in electronic information management and cybermedia, both broadly defined. He currently has two popular columns, the Digital Village in CACM, and DL Pearls, within the ACM Digital Library. He has received the ACM Distinguished Service Award, the ACM Outstanding Contribution Award, and is a Fellow of both the ACM and IEEE. http://www.acm.org/hlb.