Shaping the Software Option at the University of Alberta

Lukasz Kurgan, Scott Dick, Petr Musilek, and Marek Reformat
University of Alberta
Department of Electrical and Computer Engineering
ECERF, 2nd Floor
Edmonton, Alberta, Canada T6G 2V4
{lkurgan, dick, reform, musilek}@ece.ualberta.ca

Abstract

Teaching Software Engineering (SE) is a challenging task. There is a broad spectrum of topics which are part of the SE discipline, and it is not possible to cover all of them. What is needed is a representative set of subjects which demonstrate to students the fundamental aspects of SE and increase their understanding of the tasks and problems SE deals with. An additional difficulty is the need to overcome the students' perceptions about SE, which are substantially different from real-world practice.

The Computer Engineering - Software Option degree is a fairly new undergraduate program at the University of Alberta. The program is currently being reshaped from its original version. These modifications account for CEAB accreditation requirements, new facilities, students' feedback, and new approaches in delivering course content to the students. The paper gives an overview of the changes and our experiences in teaching SE courses by highlighting several exciting developments.

1. Introduction

The Computer Engineering - Software Option (SO) degree in the Department of Electrical and Computer Engineering at the University of Alberta was created in the late 90's, with the first cohort graduating in 2003. The program currently has about 80 students, and is supported by several full-time faculty members and two lab technicians. It is a fairly new undergraduate program with the long-term goal of delivering an industry-relevant, lab-based, and practical SE background to the SO stream students. Since its foundation, the program has undergone a steady evolution that will be finalized in the 2005-2006 academic year. The changes accommodate for a variety of factors, like accreditation requirements, new faculty and facilities, students' feedback, and new developments in the SE domain. This paper highlights several main factors that were identified to have an impact on the program, and solutions that are either currently deployed or will shortly be rolled out in the program.

In practice, teaching SE is difficult because students often do not understand the seriousness of the knowledge that is conveyed to them. We have observed that a recurring theme in the classes is a lack of belief in software engineering as a true engineering discipline. Although the SO is offered to engineering students, they appear to arrive with a preconception that software development is akin to a craft, rather than an engineering process. At the University of Alberta, undergraduate computer engineering students are first exposed to general engineering courses; and can thus be presumed to have begun developing the engineering "set of mind." Despite this, it is difficult to persuade students to apply that "set of mind" to their software engineering courses.

Our experience has been that the students' first reaction is that SE more an artistic process than an engineering one. Specific challenges that were encountered, which are similar to those described in [19], are:

1. Student's immaturity. The majority of the students have little or no exposure to industrial software development, being first-time-in-college students straight out of high school. Virtually none of the students currently in the program are non-traditional students with industrial experience.
2. Unexpected ways of doing business. Before the students take the SO courses, they complete a series of computer engineering classes, and simply expect that SE is no more then just a coding activity. They have yet to understand that it also includes many software lifecycle activities, such as size and time estimation, scheduling, tracking, requirements engineering, software design, etc.

Students are surprised to learn that merely "hacking code" will not be an adequate methodology in either the SO degree or industry.

3. Divergence between students' actual and perceived abilities. Large-scale software development requires students to reflect on their own and other developers' performance. This is difficult since students' actual abilities to write software in an organized fashion differ significantly from their perceived abilities. This raises a serious issue of students not paying enough attention to the material, thinking that it is trivial and does not require any substantial effort beyond learning how to become a master code-writer.

Contrary to the students' view, SE is defined as the "application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software" [5]. This definition stresses the importance of teaching not only software design methods and programming languages, but most of all teaching aspects such as requirements engineering, software processes, software testing and software quality, just to name a few important topics. Upon graduation students must be able to contribute to large-scale software development projects, which are characterized by well-defined and verifiable functionality and quality.

There are several mechanisms applied to deliver a high-quality SE program to the SO students, which strive to overcome the above issues:

- **Sound structure**
  The core SO classes are organized in the following manner:
  - *Introductory overview classes*: CMPE 210 and CMPE 300. The goal of these classes is to raise awareness of the problems associated with SE, and lay out background information.
  - *Specialized classes*:
    - CMPE 310 that introduces software requirements engineering;
    - CMPUT 301 that introduces software design with a focus on user interfaces;
    - CMPE 410 that introduces advanced design and programming concepts;
    - CMPE 420 that introduces construction of reliable and secure software systems.

    The goal of these classes is to expose students to advanced SE principles and methodologies. There are also plans to add a class devoted to the subject of software testing and advanced software development paradigms.
  - *A capstone project*: the program is summarized with a realistic project, which gives hands-on exposure to problems associated with development of larger software systems.
- **Project-based courses**
  SE is about working in large projects in teams, and hence projects in the education are an indispensable ingredient. The majority of the core SE classes have a lab component that enables direct exposure to concepts learned in the classroom. This is a very effective approach, which is used to familiarize the students with the "unexpected way of doing business", while at the same time teaching how to reliably assess skills and abilities.
- **Example-driven delivery of class lectures**
  The students are exposed to the class material in an example-driven way. This provides a deeper understanding of the inherent complexity and problems associated with the concepts taught.
- **Combination of research and education**
  It is essential to involve the same people, in terms of academic staff and the postgraduate students, in both research and teaching. We, as well as other researchers [21], think that in this way students will be engaged in cutting edge problems and challenges. At the same time, this ensures that the latest findings in SE are included in the courses. As a result, students graduate with an awareness of current topics and trends in the software development industry, which provides an advantage in the job market.

The remainder of the paper overviews several of the most important highlights of the SO program. The focus is to present major developments in the existing program in all the main teaching areas.

2. First Things First

The first course in the Software Option that is entirely dedicated to Software Engineering issues is *CMPE 210: Principles of Software Implementation*. The main focus of this course is put on the concepts and topics related to good programming style and development process.

The first two weeks of the course give exposure to software development principles. Such concepts as rigor and formality, separation of concerns, abstraction, and anticipation of change, modularity, generality and incrementality are introduced and explained. Next, the fundamental principle of information hiding is discussed. The benefits of using

this principle are covered, and the common techniques used to realize this principle are explained. The last topic from the Software Engineering principles is coupling and cohesion. Both measures are explained in detail and illustrated by a number of different examples. It is shown how coupling and cohesion can be used to recognize and measure the degree of component independence.

The rest of the course is dedicated to two topics, which are directly related to the software development process on the personal level. One of them is about the rules of a good programming style; the second one is about a process of programming and its improvement. Both topics are taught concurrently.

The theme of a good programming style is covered by detailed explanations of many implementation aspects related to the basic as well as advanced structures of the C programming language [11] [13]. Some of the topics covered are:

- data structure issues (usage of variables, fundamental and complex data types);
- control issues (conditionals, control loops);
- general issues (self-documented code, defensive programming and programming tools).

Additionally, many concepts related directly to software quality are covered. Students become familiar with quality attributes, a process of quality assurance, and different kinds of quality reviews such as inspections, walkthroughs and code reading. One whole week is dedicated to unit testing. A process for generating test cases for statement and branch coverage is shown and explained. Debugging is also covered. Effective approaches for finding defects together with numerous case studies are presented in the class. Many suggestions and important aspects of fixing defects are also provided.

The individual software process is taught in the framework of the Personal Software Process (PSP) [10]. The PSP is a scaled-down version of those practices described in the Capability Maturity Model for Software (CMM) that are suitable for individual use. PSP methods are based on elements of engineering, quality management and scientific disciplines which are made practical for application at the personal process level.
PSP embraces measurement of the process performance, analysis of process measurements, adjustments to and improvements of the process. The main objective is to show how to control, measure, and improve software development. A series of lectures is designed to show students how application of PSP techniques leads to improvement of the software development processes. A schedule of topics taught in these lectures is:

- introduction to PSP;
- planning overview and size measurement;
- size estimation;
- resource and schedule planning;
- process measurement;
- design and code reviews.

The main motivation behind the introduction of PSP to students is the fact that PSP helps them to improve their performance by bringing discipline to their software development process. PSP embraces measurement of the process performance, analysis of process measurements, adjustments and improvements of the process. This is a key step in breaking down the student's preconceived notion that SE is a trivial, artistic process, and inculcating the beginnings of a professional engineering approach to software development.

The course contains a lab component which allows students to practice the programming style standards discussed in class. All programming activities are performed in the framework of PSP. Students are taught, and apply, five PSP phases:

- 0 – introduces time measurement, and monitoring of defects,
- 0.1 – adds coding standard and size measurement,
- 1.0 – introduces size estimation and systematic approach to testing,
- 1.1 – task planning and schedule planning,
- 2.0 – provides basic techniques of design and code review.

3. Introduction to Software Engineering: Real World Systems

CMPE 300, Introduction to Software Development Process, is the only course in the SO lineup that is mandatory also for Computer Engineering students. It is an overview course that covers most major SE topics and introduces necessary terminology. As the Computer Engineering stream students cannot be presumed to have taken CMPE 210, the issue of student preconceptions is again crucial.

The lecture component of this course consists of three lectures per week. It starts with the introduction of software engineering as a discipline necessary to build complex software systems in context of constant change [3]. Subsequently, the Unified Modeling

Language (UML) is introduced, which is then used throughout the course to model not only objects and entities of software systems, but also to illustrate important concepts such as software processes and project deliverables. The core part of the course provides introduction to the major activities of software development, including requirements analysis, static and dynamic modeling, system design and object design, implementation, testing, and maintenance. Design and implementation concentrate on modern approaches including component based development, design patterns, and frameworks. The course also provides an introduction to some more advanced topics such as project and rationale management.

To provide exposure to real world problems, the course has been recently redesigned to include non-trivial team projects. Students form teams that undertake development of software application for a mobile robot. The course uses six Amigobots [1] equipped with ultrasonic sensors, passive grippers, and wireless Ethernet connections, as well as one color camera. These robots come with an extensive class library called ARIA (ActivMedia Robotics Interface for Applications). ARIA is a powerful API (Application Programming Interface) to ActivMedia mobile robots, usable under Linux or Windows in C++ (Java and Python wrappers are also available). With such infrastructure, students can make use of existing components and concentrate on the development process and building functionality of their systems instead of routine programming of robotic behaviors. At the same time, robots provide an application platform that is very tangible (they really move) and has real-time-real-world characteristics (latencies, speed limitations, imprecision and ambiguity of sensors). Although complicated devices, robots allow almost immediate visual assessment of the quality of design or coding. Robots are also very attractive, which results in a substantial increase of students' interest, their enthusiasm, and hence their ability to learn. The projects can be selected from a growing list, or proposed by student teams. The teams are actively involved from the project inception to the product delivery and public presentation.

During the two term history of this updated course we have already seen many ingenious application ideas and high quality software development projects. The project topics range from simple demonstrations of common driving procedures (backing up, parallel parking) to advanced applications (3D mapping, line following, support for visually impaired).

4. Software Requirements Engineering: A Hands-On Approach

The *CMPE 310* class is devoted to exposing students to the area of software requirements engineering. The class provides a comprehensive review of widely used industrial practices for the elicitation, documentation, modeling, and validation of requirements. While this subject is often discussed only in introductory SE classes, our observations of the inability of students to understand the importance of, and the difficulty inherent in, developing a correct, consistent, and complete software requirement specification (SRS) document led to the development of this class. This relates to all three challenges listed in section 1. In particular, students (who at the point of taking this class are already exposed to UML) often think that use case diagrams is all that they need to properly understand and document software requirements.

The two main themes of the class are: 1. "elicit, do not assume", and 2. "structured approach and formalisms pay off". Those two themes are used to define the structure of the class, which consists of two half-semester blocks. The class consists of lectures, which are strongly supported by the lab components, where students gain hands-on experience with the concepts.

The first half is driven by the first theme, where students are exposed to the software requirements engineering process, several elicitation techniques, the software vision document, and the SRS document standards. The specific topics include

- Exposure to basic software requirements engineering techniques by applying a mixture of in-class lectures, readings, and student presentations on a number of relevant publications [2] [8] [9] [14] [17] [18]. The publications are carefully selected to describe all main components of the software requirements engineering process, and at the same time challenge the students with self-understanding of the material combined with in-lab presentation. The readings and presentations are performed in groups of 2-3 students, and are also used to reinforce communication skills.
- Elicitation techniques, such as interviewing, workshops, use case driven elicitation, and software prototyping. There are two labs devoted to the elicitation process, where the students learn to elicit a software system in an interactive manner.

The instructor's role is to simulate a customer, while the students learn hands-on how to elicit information, and how to communicate and refine understanding of the problem. We note that for most of the students this is the first contact with elicitation techniques, and such hands-on exposure strongly reinforces their understanding of the complexity of this task. An emphasis is also placed on use case driven elicitation being much more than the development of use case diagrams. The students are exposed to, and expected to work in the lab with, the textual use case standards that provide much more information than diagrams.

- Exposure to several industrial standards for developing both the vision and the SRS documents, such as the IEEE 830 standard [4]. The students end the first part of the class developing a larger SRS document in the IEEE 830 standard.

The content of the first half of the class is supported by a textbook by Wiegers [20], which provides a practical, industry-influenced view of software requirements engineering principles and elicitation techniques.

The second half of the semester is driven by the second theme. The students are exposed to several formal software requirement engineering techniques, with the goal to apply them in a real-life context. The specific topics include:

- Operational specification models, such as Finite State Machines and Petri Net models,
- Descriptive specification models, such as algebraic specifications, and Rigorous Approach to Industrial Software Engineering (RAISE) model [15] [16],
- Other formal methodologies, such as design by contract,
- and the concepts and methods of verification and validation of the requirements.

The content of the second half of the class is supported by a textbook by Ghezzi et al. [6], which provides an excellent, example-driven introduction to the relevant formal methodologies. A strong emphasis is put on applying the learned concept in practice by providing hands-on experience during the labs. The students are asked to model a real-life system using the above methodologies, and with the goal of learning their benefits and downsides.

An indispensable feature of the class is its project driven theme. Over the course of the entire semester the students are asked to elicit, document, and analyze requirements of a real-life software system. The task is divided into five project assignments that concern a system for controlling traffic lights at a predefined road intersection. The choice of the system is driven by the two main themes of the class. First, the students learn that only a proper elicitation process leads to correct understanding of the system. The goal is to model a well-know physical system, which is subject to customer imposed assumptions that result in a different-than-expected behavior pattern. Second, the students apply formal techniques to properly analyze the system.

The current design of the class resulted in positive student reactions, who appreciate exposure to learning the details of the material through self-reading, presentations, project, and interactive lab sessions.

5. M-Commerce in the Capstone Project

A recent development in the Software Option is the incorporation of a capstone design experience. As a part of the CEAB accreditation requirements for software engineering, *CMPE 440 Software Systems Design Project* was rolled-out in Winter 2003, providing a capstone design experience for the first cohort of Software Option students. In common with other capstone design courses, students are required to work in teams to conceive, design and implement a reasonably large software system. Team members are randomly assigned, avoiding self-selection of homogeneous teams [7]. These teams must then promptly select from a list of proposed project topics, or else propose one of their own. The teams must prepare requirements documents (which require the instructor's approval before design may begin), design documents and a working implementation of their software system, using the skills and knowledge gained during the course of their education. Team skills are vital, as the students must adjust to an unexpected group of teammates and meet the absolute requirement of a working software system, all within the 13-week semester.

The CMPE 440 course is taught as a single lecture hour each week, coupled with 6 hours of lab time. During the semester, three formal laboratories are used to introduce the students to industrial-grade software development tools; as of this writing, these tools include the CVS version control system; the Bugzilla defect-tracking tool; and the JUnit unit-testing framework. A laboratory on refactoring, using the Eclipse IDE, is being considered for roll-out in Winter 2005.

A principal development in Winter 2004 is the introduction of *mobile computing* and *m-commerce* as project areas for the students to work on. This requires both the introduction of mobile computing concepts (such as context awareness, network and access-point handovers, and security and privacy concerns), and the provision of a mobile-computing infrastructure for student projects. This infrastructure is being implemented via a Bluetooth wireless LAN; Bluetooth was selected over 802.11 technologies because of the limited space available in the Software Engineering laboratory. Limited signal ranges are a signature feature of mobile computing, and the signal range of 802.11 technologies was simply too large to allow students to experiment with this phenomenon. Instead, the Bluetooth infrastructure consists of Bluetooth-equipped PDAs running PalmOS and J2ME, and Bluetooth access points that will tie into the existing wired LAN in the Software Engineering lab via DHCP. Providing 3 access points and six shared PDAs (to be augmented with a further 4 PDAs in Winter 2005) will allow the students to simulate a variety of m-computing scenarios; the total cost of this infrastructure, on the other hand, will be roughly $8000 over the two-year period. Student interest in the m-computing projects has been very high; of 8 teams in Winter 2004, 3 have elected m-computing or m-commerce projects. The students in this term were also given the opportunity to help define the m-computing infrastructure for the course; their suggestions were well thought-out and helped determine the ultimate direction for the Bluetooth infrastructure. This first round of m-computing projects will be completed by the end of March 2004, and we eagerly await the presentations by these teams.

6. Summary and Future Work

To summarize, the Software Option at University of Alberta is becoming a mature undergraduate program with many exciting and strong cornerstones. However the current model is still being redefined and upgraded.

There are two main directions envisioned for the future. First, there is a strong support for industrial involvement, by means of the approaches described in [21]. For example, the software requirements engineering class evolves towards involvement of industrials partners, who will act as customers in the elicitation and SRS development process. Naturally, these industrial partners would also make ideal clients for teams in the capstone design experience, although care must be taken that no student project falls on the critical path of any partner's development efforts.

Secondly, there is a need for an increased emphasis on management and business issues that are related to the SE domain. It is well-known that political and organizational considerations are key determinants of the success of a software project; as future software engineers, students must be exposed to these considerations. A good overview of possible avenues to explore is given in [12].

References

[1] Amogobot website, http://www.amigobot.com/

[2] Austin, R., and Devin, L., Beyond Requirements: Software Making as Art, *IEEE Software*, pp. 93-95, Jan/Feb 2003

[3] Bruegge, B., Dutoit, A. H., *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall, 2004

[4] IEEE Standard 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE-SA Standards Board, 1998

[5] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Standard 610.12, 1990

[6] Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2003

[7] Goold, A., and Horan, P., Foundation Software Engineering Practices for Capstone Projects and Beyond, *Proceedings of the 15th Conference on Software Engineering Education and Training*, pp. 140-147, 2002

[8] Gottesdiener, E., Requirements by Collaboration: Getting It Right the First Time, *IEEE Software*, pp. 52-55, Mar/Apr 2003

[9] Graham, D., Requirements and Testing: Seven Missing-Link Myths, *IEEE Software*, pp. 15-17, Sep/Oct 2002

[10] Humphrey, W.S., *A Discipline for Software Engineering*, SEI Series in Software Engineering, Addison-Wesley, 1995

[11] Hunt, A., and Thomas, D., *The Pragmatic Programmer*, Addison-Wesley, 2000

[12] Lawrence, P., Educating Software Engineering Managers, *Proceedings of the 16th Conference on Software Engineering Education and Training*, pp.78-85, 2003

[13] McConnell, S., *Code Complete*, Microsoft Press, 1993

[14] Neill, C., and Laplante, P., Requirements Engineering: The State of the Practice, *IEEE Software*, pp. 40-45, Nov/Dec 2003

[15] RAISE Method Group, *The RAISE Development Method*, BCS Practitioner Series, Prentice Hall, 1995

[16] RAISE Language Group, *The RAISE Specification Language,* BCS Practitioner Series, Prentice Hall, 1992

[17] Rupp, C., Requirements and Psychology, *IEEE Software*, pp. 16-18, May/Jun 2002

[18] Tveito, A., and Hasvold, P., Requirements in the Medical Domain: Experiences and Prescriptions, *IEEE Software*, pp. 66-69, Nov/Dec 2002

[19] Umphress, D., and Hamilton, J. Jr., Software Process as a Foundation for Teaching, Learning, and Accrediting, *Proceedings of the 15th Conference on Software Engineering Education and Training*, pp.160-169, 2002

[20] Wiegers, K., *Software Requirements*, 2nd Edition, Microsoft Press, 2003

[21] Wohlin, C., and Regnell, B., Achieving Industrial Relevance in Software Engineering Education, *Proceedings of the 12th Conference on Software Engineering Education and Training*, pp. 16-25, 1999