An Undergraduate Option in Software Engineering: Analysis and Rationale

Timothy C. Lethbridge[σ], Dan Ionescu[ε], Ali Mili[σ], David Gibbons[ε]
[σ]Department of Computer Science
[ε]Department of Electrical and Computer Engineering
University of Ottawa
Ottawa, ON, K1N 6N5, Canada

Abstract

This paper discusses an undergraduate option in software engineering which is currently in the approval process at the University of Ottawa. The proposed option represents a close collaboration between the Department of Computer Science and the Department of Electrical and Computer Engineering. In this paper we present an analysis of the weaknesses of the graduates of existing programs and some criteria for the choice of courses in the new option. In particular, we examine why a significant amount of additional project-based experience should be required in the software engineering option. The resulting option differs from those at other universities in that it has a somewhat greater emphasis on both human factors and the object oriented approach.

1. Introduction

At the University of Ottawa, we are in the process of establishing an option in software engineering. The option will be coordinated jointly by the Department of Computer Science and the Department of Electrical and Computer Engineering. Although details are still to be finalized, the goal is to initiate the option in the 1997-98 academic year.

In this paper, we describe the option and present its rationale. The driving force behind design of the option has been the needs of graduates as they enter software engineering careers in industry. Our main resources for background information have been:

- Other undergraduate software engineering programs and options [1], [2], [3].
- Discussions with software practitioners and industrial managers to determine the knowledge and skills that appear to be most lacking.

The need to educate more well trained software engineers is an international problem. However it is particularly acute in Ottawa, where rapidly growing local industries estimate they need over 4000 new software professionals a year [4].

In section 2, we discuss perceived advantages and disadvantages of current programs in Computer Science and Computer Engineering (CS/CE). In section 3, we discuss why software engineering skills should be taught in undergraduate programs at universities, rather than in industry. In section 4, we outline the criteria we used for designing courses for the option. Then finally, in section 5, we present a summary of the option as currently proposed.

2. The status quo: Strengths and weaknesses of existing programs

We have discussed CS/CE programs with a significant number of industrial managers. These discussions have led us to formulate the following lists of strengths and weaknesses of existing university programs in many North American universities.

2.1 Strengths of existing programs

Current CS/CE programs produce students who have good knowledge in the following areas. Such strengths should be preserved as more software engineering principles are introduced into the curriculum.

- Mathematics and the theoretical background useful in software engineering.
- Data structures and algorithms.
- The environment in which software engineering is performed, and the tools which are used (e.g. programming languages and compilers; operating systems; databases, etc.).

In addition, many (but not all) programs give students good exposure to other areas of knowledge (e.g. business, other sciences, electronics, etc.). This cross-disciplinary material is invaluable for students who must engineer specific classes of applications.

2.2 Weaknesses of existing programs

Discussions with those in industry indicate that current CS/CE students have certain significant deficits in their education. The following are the most frequently identified. We have designed our software engineering option with the specific goal of addressing these weaknesses.

Weakness 1: Lack of opportunity to build up skills and experience in areas of software engineering that are only taught at a theoretical or overview level.

Students are typically given an overview of many software engineering techniques, but they are rarely given the opportunity to practice them in realistic, but controlled environments (i.e. on industrial-size systems, but with the guidance of a professor). Also, they are only exposed to the very basic elements of a few methodologies and techniques – they generally do not learn any technique in depth. The following are classes of skills where expertise appears particularly lacking:

a) Requirements analysis: Usually, students are given already-prepared requirements for systems they are to develop in university courses. In some senior courses they may be given limited exposure to gathering requirements, but they are rarely given projects where they must use a variety of techniques to elicit changing requirements from elusive customers, in a climate of uncertainty.

b) Specification: Students are often taught the basics of several formal or semi-formal specification techniques, but they are rarely asked to specify a complex real-world software artifact using these techniques.

c) Design, especially object oriented design and design of large systems: Students are typically taught about qualities of a good design, and they design a significant number of small programs. However, they are not generally exposed to complex design tasks (including large-system architecture) where they can be coached through classes of difficulties and learn the pros and cons of various approaches. i.e. they are not taught in the manner that is common in schools of architecture.

d) Software quality assurance: Students are rarely asked to systematically measure and critique the quality of significant systems.

e) Software project management: Although most software engineers in industry at some point manage or co-ordinate teams of software professionals, few university programs require that students develop significant skills in this area. Most programs teach the basics of project planning and cost estimation, but more in-depth studies are either absent or else relegated to an elective. Students are rarely given the kinds of case-studies common in business schools.

Each of the above areas corresponds to a major source of project failure. An influx of effectively educated software engineers would therefore be a significant benefit to industry. It is our contention that if students were given the opportunity to work on projects in each of these areas, at the same time as they learn a methodology in depth, they will be much better prepared to perform software engineering tasks.

Weakness 2: Lack of exposure to large systems.

Topics discussed under weakness 1 are usually briefly covered at least at a theoretical level. However, there are several areas of great importance to software engineering that most computer science and computer engineering programs omit almost entirely. One of these is the development and evolution of large systems. Underrepresented subtopics include:

- Configuration management
- Reliability engineering
- Release management
- Reengineering
- System integration.

Weakness 3: Lack of exposure to human factors.

A second area where students are not always even given an overview is human factors. This is regrettable given the fact that the software industry is labor intensive and produces complex and changing products that must be used by people. A common estimate is that over 50 percent of the cost of computer projects can be attributed to the user interface [5], and in some projects this fraction can be 70 percent or more.

Courses in human-computer interaction exist at many institutions. These cover such topics as the psychology of users, usability engineering, evaluation of interfaces and user interface design. Unfortunately these courses are almost always electives, taken by a small subset of senior students. Even the SEI undergraduate curriculum [3] and the ACM/IEEE-CS curriculum [6] propose human factors as only a small component of one course.

Our observations and discussions in industry have shown us that human factors is an area where not only is the material not known, but that managers and software engineers *do not know* that they do not know it. i.e. they produce products with low usability, and do not realize that they could have done much better.

Weakness 4: Lack of exposure to education about business and communication.

When we presented the first version of our option to a group of senior industrial managers, they liked the proposal in general, but they emphasized one additional weakness that we had to overcome: The need for more ‘application’ and ‘professional’ skills, particularly in business administration and writing. Of particular concern here are the following observations:

- A high percentage of what software engineers do is to write documents (proposals, requirements etc.). Yet many do not have adequate writing skills. Few university computer courses that require writing put emphasis on the quality of that writing; and students are only required to take a small number of courses in subjects where writing is valued more highly.
- Most software development is done to support business activities; hence knowledge of finance, management and accounting will greatly facilitate the software engineering process.. However, the number of business courses taken by students varies from program to program. In many universities, such courses are optional and their importance is not communicated to students..

3. Two approaches to software engineering education: university-based training vs. employer-based training

Most would agree that university courses are the appropriate place to address weaknesses 2, 3 and 4. However, there is disagreement about weakness 1. We encountered people opposed to increasing the practical content of university-based software engineering education. This section discusses their arguments and then presents the counter-reasoning we have used to justify certain decisions in our option.

The following are a few of the arguments that have been made for leaving in-depth training and exposure to large practical problems until a students starts to work in industry:

Argument A: Traditionally, students in engineering programs continue their apprenticeship by learning about practical matters following graduation.

Argument B: Companies want to teach students their own in-house methods.

Argument C: There is not time in the university curriculum for everything; and so the university had better focus on theory, which will almost certainly not be taught in industry.

Argument D: There may not be adequate resources in the university, especially professors with strong backgrounds in software engineering.

However, the following counter-arguments can be made that favor moving more practical material and industrial examples into university software engineering courses:

Counter-arguments to argument A (tradition):

1. In other areas of science and engineering, techniques that were once taught in industry or graduate school are now being taught to undergraduates.
2. The following three problems (a to c) suggest that it should be more *efficient* to teach students in a well-planned academic environment than in an industrial environment (where education is not the main goal). In industrial environments, students learn in a largely ad-hoc manner, through experience and mentoring, supplemented by a few overview courses.
   a) Ad-hoc mentoring consumes the time of experienced engineers in an unpredictable way and can disrupt their work.
   b) Industry-based education tends to be one-to-one and is thus repeated for multiple learners. Furthermore, there is no written record of the education.
   c) Such education relies on the willingness of the learner to experiment or ask questions. Many will just proceed in an ineffective manner, not knowing that there are better methods.

3. Academic courses can ensure that students are given good coverage of the latest techniques; industrial training is likely to cover only a subset.
4. In an academic environment, students can experiment with techniques without dire consequences if they fail.
5. Students who are given more exposure to the practicalities of real systems will be more likely to see the relevance of the theoretical material covered later.

Counter-argument to argument B (in-house methods): Students who are only taught what a given company wants will be less able to introduce new ideas into a company.

Counter-argument to argument C (no time): We believe this is a matter of better course design. Courses can be designed around case studies (as is done in business school), design coaching (as in architecture school) and direct collaboration with industry (so students can perform analysis and design on real industrial problems).

Counter-argument to argument D (no resources): This is the hardest point to refute, since skilled software engineering professors would be paid much more if they were to move to industry. We believe the solution is for industry to actively invest in teaching in universities (e.g. by establishing chairs in software engineering education, giving special funds to boost curriculum development, etc.).

4. Criteria for Design of a Software Engineering Option

In this section we crystallize the discussions from sections 2 and 3. We do this by presenting criteria for the academic content of senior courses we are proposing for the University of Ottawa's software engineering option.

Criterion 1: Consistent with other programs. The program as a whole should cover most or all of the material considered important in software engineering curricula at other universities. We looked, in particular, at curricula proposed by the Software Engineering Institute [3] and the University of Waterloo [2]. However, the other criteria below led to some differences from these programs; these are discussed in section 5.4. We also considered the ACM/IEEE-CS computer science curriculum [6]: We found that our software engineering option can be based on it in large measure, with specialization in the last two years.

Criterion 2. Responds to industrial needs. The courses as a group should ensure that coverage and weighting of material reflect the current and future needs of practicing software engineers, with particular regard to helping them solve problems they will encounter in their work. The weaknesses discussed in section 2 should be addressed.

Criterion 3. Self-contained courses: Each course should be self contained so that practitioners from industry will be encouraged to take it. Such people might not be following the entire program, but be in need of a certain body of knowledge. By self-contained we mean that the course should have a unifying theme with only a few general prerequisites.

Criterion 4: Focus and depth in each course. Where many methodologies exist, it would be preferable to cover one in depth so students have a marketable skill, rather than an overview of many. At the same time, students should at least be given an awareness of those techniques that are not studied in depth.

Criterion 5: Broad applicability of compulsory courses. The compulsory material should be useful to most or all software engineers, regardless of the kinds of systems they will be engineering.

Criterion 6: Projects and practical examples in senior courses. There should be projects in as many courses as possible, and these should involve large industrial systems. Students should be coached through the process of doing analysis or design in the context of these systems. In non-project parts of courses, there should also be liberal use of case studies from industry.

5. Outline of the proposed option

In this section we present the University of Ottawa's proposed option in software engineering and explain our choices based on the issues raised in sections 2 and 4. Figure 1 gives a schematic overview of the option. The authors will provide full details on request.

5.1 The first three years.

The first three years of our option follow a plan that is very similar to many existing computer science or computer engineering programs. In these years, both CS and CE students will take many common courses (see figure 1), but they will also cover certain common material in *separate* courses (for administrative reasons, and to mesh with other requirements of the programs). In particular, the latter applies to data structures, algorithms and programming languages.

Among the common courses are two intensified courses in programming to be taught at the second year level. These will ensure that students entering software engineering courses have a very strong grounding in programming.

Computer science students taking the option will have the opportunity to take courses in business administration or other fields to help satisfy weakness 4. Computer engineers will follow the common curriculum established for all engineering students in the first year, and will take additional courses in computer hardware and related engineering topics.

5.2 The four central 4th-year courses.

This subsection describes the four 4th-year courses that will be made compulsory as part of the proposed option. These courses have the following general features:

- They are designed to rectify weaknesses identified in section 2; in particular, they all have projects (weakness 1, criterion 6), and they all provide skills that are in need in industry (criterion 2).
- They are self contained (criterion 3) so that special students from industry can take then even if not following a program.

Taken as a whole, the courses cover material that we believe every software engineer will find important (criterion 5). The courses also cover much of the material in other software engineering options or programs (criterion 1).

For each course, we present the proposed calendar description, followed by some additional comments.

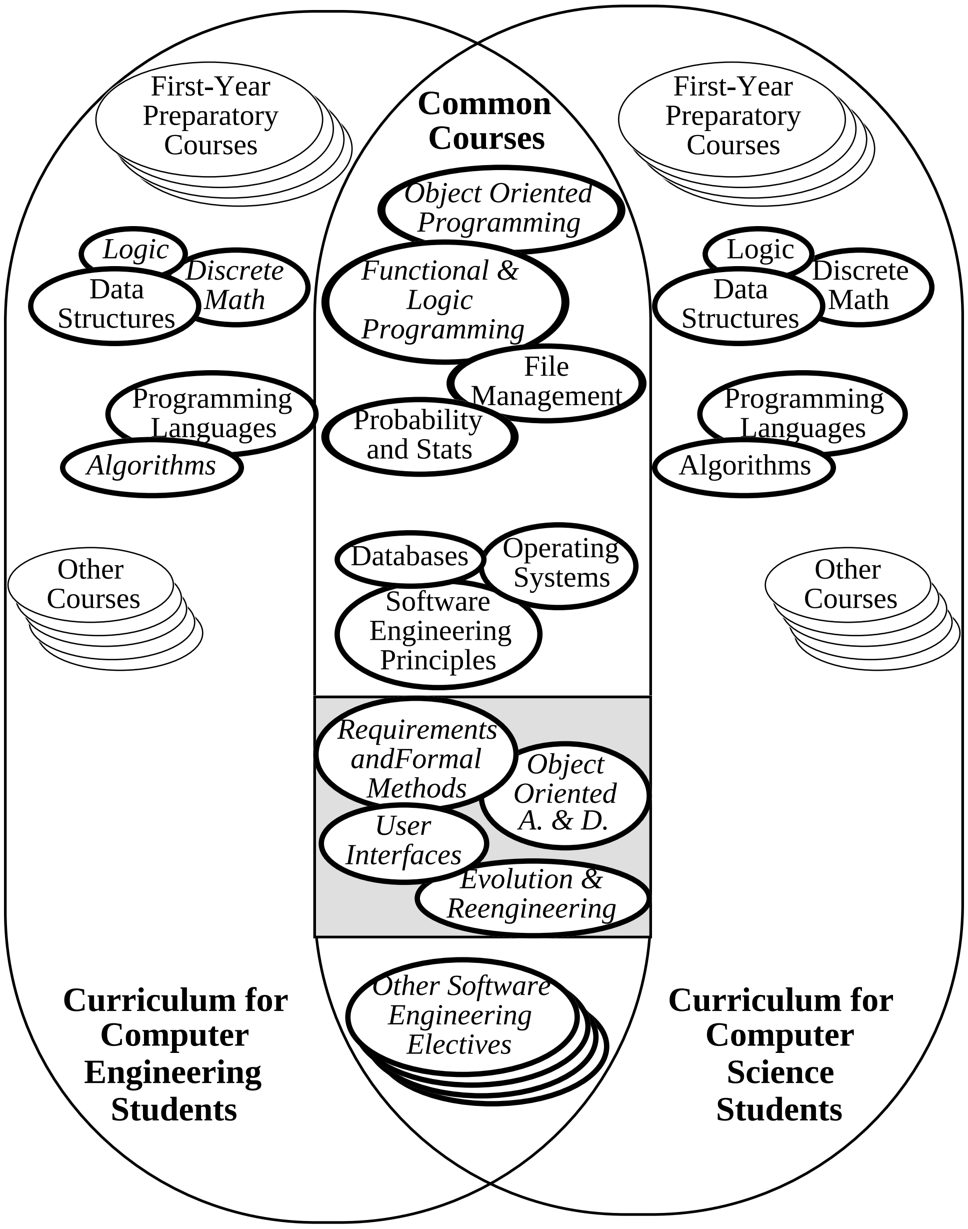
First-Year Preparatory Courses
Common Courses
First-Year Preparatory Courses
Object Oriented Programming
Logic
Discrete Math
Data Structures
Functional & Logic Programming
Logic
Discrete Math
Data Structures
File Management
Programming Languages
Probability and Stats
Programming Languages
Algorithms
Algorithms
Databases
Operating Systems
Other Courses
Software Engineering Principles
Other Courses
Requirements andFormal Methods
Object Oriented A. & D.
User Interfaces
Evolution & Reengineering
Other Software Engineering Electives
Curriculum for Computer Engineering Students
Curriculum for Computer Science Students

Figure 1. Summary of material in the proposed University of Ottawa Option in Software Engineering. Bold ellipses represent common material for all students in the option; those to the left and right are covered in slightly different ways depending on department. In the centre are courses all students in the option will take. Italics indicate new and modified material for the option. The shaded region contains the central fourth-year courses. Note: The labels are not official course names, but instead they are short descriptive labels.

Formal Methods in Software Engineering: Requirements gathering techniques. Formal approaches to specification of requirements. Use of a specification language such as Z. Certification and validation techniques. Zero defect software technology. Requirements engineering. Using CASE tools to model requirements. Handling uncertain and changing requirements. Formal development project.

This course covers requirements, and their specification using formal techniques. The need for better knowledge about requirements analysis is well acknowledged in industry (weaknesses 1a and 1b). The formal methods material prepares students for the day when software engineering is a more precise discipline – graduating students will hopefully help industry to assimilate these ideas.

Analysis and Design of User Interfaces: Psychological principles of human-computer interaction. Evaluation of user interfaces. Usability engineering. Task analysis and user-centered design. Conceptual models and metaphors. Prototyping and iterative design. Software design rationale. Screen, menu and command design. Interface modalities: Color and sound. Internationalization and localization. User interface APIs. Case studies and project.

This course responds to weakness 3. Few programs make such a course mandatory for software engineers; however ,we feel that it is one of the more essential subjects, and is applicable to almost all software. Preparing a report about the usability of an existing system will provide opportunities to practice communications skills (weakness 4).

Object Oriented Analysis and Design: Object oriented analysis. Object oriented design. Frameworks. Reuse. Advanced object oriented programming techniques. Object oriented databases. Comparison of object oriented and non-object oriented methodologies. Project involving object oriented analysis, design and implementation.

The above course responds to weaknesses 1a and 1c (the need for more intensive analysis and design experience). An alternative to would have been to teach a more general course that covers a variety of advanced analysis and design techniques. However, we focus on the object oriented approach alone here because:

- We want to promote in-depth learning (criterion 4).
- It is generally applicable to a wide variety of problems (including real time systems) (criterion 5).
- It is in particular demand in industry and by students.
- To teach it well requires a full course.

Students will still be exposed to functional analysis and design in the third-year introduction to software engineering. Real time system design, on the other hand, is an elective in computer science but compulsory in computer engineering.

Software Evolution and Reengineering: Maintainability and reusability analysis. Approaches to maintenance and long-term software development. Change management and impact analysis. Release and configuration management. Software metrics. Advanced testing techniques. Software process standards. Reengineering and reverse engineering. Statistical software reliability engineering. Case studies and project in analysis and maintenance of existing large systems.

This course responds to weaknesses 2 and 1d. It would also be a good stand-alone course for those in industry who are developing or managing large and/or legacy systems (criterion 3). This is another course where students will be required to practice writing and communication as they present an analysis of an existing system (weakness 4).

5.3 Software engineering electives

In the proposed option, students from both departments may take each other's software engineering electives (which are all 4th year courses). Several new such courses have been added. A couple of particularly important ones are described below.

Software Engineering Project: An integrated program of lectures by invited speakers active in software engineering, and the preparation and presentation of group projects under the guidance of academic staff. Project selection is done in the early fall. At specific dates during the year, students are required to submit: a) A project plan with a cost estimation; b) Specifications; c) A design and implementation report, and d) A quality assurance report.

For computer science students (but not for computer engineering students), this will actually be compulsory and will replace the traditional project course which had more relaxed requirements. This course adds to the practical work that students will already have accomplished through the courses listed in section 5.2.

Software Project Management: Software project planning and scheduling. Organizing and managing software teams. Monitoring and controlling software development. Cost estimation techniques including function point counting. Factors influencing productivity and success. Analysis of options and risk. Planning for change. Managing expectations. Software contracts and intellectual property. Ethics and professionalism in software engineering. Case studies. Project involving forensic project management of a real industrial project.

This elective course responds to weakness 1e and, to a limited extent, weakness 4. It builds on material learned in the third year introduction to software engineering.

5.4 How does our proposed option compare with other software engineering programs?

The biggest differences with respect to programs or options at other universities are:

- A greater focus on human factors (a complete compulsory course).
- Our choice to cover object oriented analysis and design in depth rather than cover a variety of analysis techniques in a more general manner. Students will still receive an overview of other techniques.
- The manner in which we have integrated several significant projects involving industrial software, including two that analyze existing systems and allow students to practice effective communications skills.
- The way we have integrated the option into Computer Science and Computer Engineering using a great many common courses.

5.5 How does our option fit within the ACM/IEEE-CS computer science curriculum?

The fourth-year courses described above build on many areas of the ACM/IEEE-CS curriculum [6], but particularly the five SE (Software Methodology and Engineering) knowledge units. At the University of Ottawa, these units are taken in the third-year software engineering course. The user interface course also builds on HU1 (User Interfaces).

6. Concluding Remarks

We have presented the rationale behind the proposed University of Ottawa option in Software Engineering. The strengths of this option are:

• It responds to the needs of industry, while maintaining a base of accepted academic content. Such needs include improved coverage of requirements analysis, human factors, project management, analysis and design (especially of large systems). Also considered are the needs of the practitioner who wishes to take particular courses to upgrade his or her knowledge.

• It emphasizes the importance of giving students a heavy exposure to industrial problems. It proposes following the business-school model to emphasize case studies, and to follow the architecture school model to emphasize design projects and coaching.

As of December 1996, the options discussed in this paper are still being discussed by the university. Although not yet officially approved, plans are being developed to form a new administrative structure called SITE (School of Information Technology and Engineering) within the Faculty of Engineering. SITE would be responsible for undergraduate and graduate education in software engineering, computer science, computer engineering, telecommunications and related subjects. It would result in the integration of two existing departments, namely Computer Science and Electrical and Computer Engineering. This administrative restructuring may mean that the final form of the option may differ slightly from what has been described in this paper.

Acknowledgments

The authors would like to thank the reviewers of this paper, and others who provided insightful comments or contributed to early drafts of the option: Stan Szpakowicz, Luigi Logrippo, Emil Petriu and Bob Probert.

References

1. G. Ford, (1994) *A Progress Report on Undergraduate Software Engineering Education,* CMU/SEI-94-TR-11.
2. J.M. Atlee et al., (1996) “A Joint CS/E&CE Undergraduate Option in Software Engineering”, *Proc. Conf. on Software Engineering Education*.
3. G. Ford, (1996) “THE SEI Undergraduate Curriculum in Software Engineering”.
4. P. Fillmore, “Information Technology Job Training: A Survey of Ottawa Region IT Training Institutions”, Internal Report, Ottawa-Carleton Research Institute.
5. B.A. Myers. & M.B. Rosson (1992) “Survey on user interface programming” In: Bauersfeld, P.; Bennett, J. & Lynch, G. (Eds.) *Proc. CHI'92*. Reading, MA: Addison-Wesley
6. ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. Feb 1991