A Proposed Curriculum for an Undergraduate Software Engineering Degree

Michael McCracken, Idris Hsi, Heather Richter,
Robert Waters, and Laura Burkhart
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{mike, idris, hrichter, watersr, laura}@cc.gatech.edu

Abstract

We have developed a curriculum for a software engineering undergraduate degree. We used the medical school clinical model to guide our design as it successfully combines both knowledge and practice components. Through rotations, our curriculum will provide graduates with both an advanced knowledge of software engineering concepts and practical skills that have been honed in a realistic setting. In this paper, we present our proposed curriculum and the difficulties we foresee in implementing it.

1. Introduction

Over the last 30 years, software engineering has been a growing and evolving discipline. Similarly, software engineering education has grown from a few general courses to separate specializations and degrees. The recent push to professionalize the field has stimulated the creation of undergraduate degree programs in software engineering. Recently, a joint ACM/IEEE committee was established to address accreditation of undergraduate software engineering curricula [3], and another committee has been exploring guidelines for software engineering education [2]. The next step is to explore model curriculum for software engineering undergraduate programs.

Several universities already have established an undergraduate curriculum for a bachelor's degree in software engineering, such as Rochester Institute of Technology (RIT) in the U.S. and University of Melbourne in Australia. At Georgia Tech, we are proposing a radical change for a curriculum of a software engineering undergraduate degree. Instead of providing additional software engineering courses within the typical computer science model, we propose a rotation model derived from the medical profession. Our objectives in this curriculum are to provide graduates with both an advanced knowledge of software engineering concepts and practical skills that have been honed in a realistic software development setting. We call our proposed implementation of this model the Software Factory. Within the Software Factory students will work on multiple real projects while learning and practicing key software engineering concepts. Our Software Factory is different from other programs of similar name in that it will be an extensive and immersive environment with many ongoing and parallel development activities and all the management difficulties that involves.

We believe that this kind of immersive rotation model is necessary to provide adequate training for software engineers. We are currently exploring the many issues involved in implementing our Software Factory as an undergraduate software engineering degree

program. Section 2 describes the knowledge that a software engineering student should have on completing the program and how we arrived at that content[1]. Section 3 introduces our notion of rotation and compares this to other disciplines and laboratories. Section 4 summarizes the particular rotations and their role in the curriculum. Section 5 outlines the issues that need to be considered in order implement the rotation model. Section 6 then discusses additional difficulties we foresee with this model and how we hope to overcome them.

2. Background

Software engineers design products in a manner similar to architects and engineers. Though the term software engineering is used to describe many activities, there is a common sentiment that it relies on many of the methods that engineers (e.g., mechanical or electrical) rely on to produce quality products. Of concern to us, is that educators will perceive that to be a software engineer, you must be educated as an engineer. Though the rigor and foundational attributes of an engineering education are important, our concern is with how engineers are taught; in particular, how they are taught to design. We don't feel that blindly following a model of engineering education for software engineering will produce effective practitioners. Rather, we feel engineering has ignored other successful forms of education, both in the design arena, such as architecture, and in professional schools, such as medicine. Architecture and medicine for example, recognize the need for immersive practice as a major element of education. We should note that we are aware of the numerous projects and programs that are seriously revisiting engineering education, and many of the innovative programs that are in place or being proposed at engineering schools throughout the world. Our goal was to develop a program that included an understanding of what a software engineer needs to know, as well as an understanding of how it should be taught.

We began exploring a software engineering undergraduate curriculum in a seminar of graduate software engineering students. We started with an informal analysis of the knowledge that we expected that a graduate with a bachelor's in software engineering should know. We next examined how this knowledge would be taught.

We developed the content of our curriculum in three steps. Our first step was to develop a list of knowledge units[2]that software engineering students should learn in the program. We divided these units into three major categories: basic computer science, basic software engineering, and advanced software engineering. We felt that the basic computer science and basic software engineering topics were needed for students to do "programming in the small."[3] This means that the student is able to analyze, design, code, and test small programs individually or as part of a small group. We believe students need to be skillful at this individual programming before learning advanced software engineering which involves realistic large-scale projects, or "programming in the large". Next we assigned goal levels for the knowledge units in a form similar to [1].

---

[1] We recognize the work of the Software Engineering Body of Knowledge Project. When that project is complete, we will review our recommendations to ensure they are commensurate with their recommendations.

[2] Knowledge units is the terminology we used to define topics that a student should learn — both in computer science and software engineering — before graduation.

[3] We use "programming in the small" to describe software development activities involved in developing small programs individually. Conversely, "programming in the large" means the activities involved in developing large-scale software, such as complex design, integration, quality testing, and management of activities between a number of people.

Table 1 summarizes these goal levels. Each level defines a goal of understanding as well as corresponding example methods of delivery and assessment.

Table 1 - Goal levels

Level	Goal	Method of delivery	Method of assessment
1	Awareness	Lecture, reading	Exam (fill-in-the-blank, matching, etc.)
2	Literacy	Same as previous level	Structured practice, homework, detailed exam
3	Concept and use thereof	Same as previous levels, plus well-structured project	Same as previous level, plus project performance
4	Detailed understanding and application	Same as previous levels, plus ill-structured project	Same as previous level, plus process performance
5	Skilled use	Student-directed project, independent research	Thesis/research

Finally, we assigned a percentage value to distinguish between the amount of theory versus practice required to adequately learn each knowledge unit. The theory component would be learned in a classroom environment, whereas the practice component would be done in projects and laboratories. Table 2 shows an example of several design knowledge units with the corresponding goal level and practice percentage. The basic computer science and basic software engineering knowledge units are summarized in another paper [6]. The advanced software engineering knowledge units are available in Appendix A.

Table 2 - Knowledge nuggets, goal level and practice (%)

Design	Goal level	Practice (%)
Software architecture	4	60
Real time systems	2	0
Information systems	2	0
Distributed systems	3	60
Refinement	4	95

We realized that the first two years of the new Georgia Tech computer science curriculum would satisfy our knowledge units for basic computer science and basic software engineering. Thus, the first two years of the software engineering program could take advantage of an already existing foundational core [6] allowing us to focus only on developing the final two years of the software engineering degree. The rest of this paper involves our proposal for teaching the advanced software engineering topics in these final two years. We abandoned the traditional model of just creating additional courses in software engineering and instead looked to other models of professional education.

3. Rotations

Professional education generally relies on a practice-based component as one of the major elements of learning. Medical schools use clinical models where students

participate in the diagnosis and treatment of patients in supervised settings. Architecture uses the studio model of design practice. Law schools use mock trials. Software engineering education, in our opinion, requires a practice-based component similar to the professional schools mentioned above. The debate centers on what type of practice and how much practice, not whether practice in some form is needed. The use of capstone courses as the design component of engineering education is under attack by reformers of engineering education [4]. Our own research at Georgia Tech of the effectiveness of capstone courses supports these criticisms. Our research supports our intuition that students abandon their skills and knowledge when suddenly confronted with a real design problem and muddle their way through the problem [5]. They attempt to give the answer they think the professor wants rather than solving the problem as a designer would. In our opinion, an immersive practice element is the mechanism to support the use of previously learned skills and knowledge and to build on that learning by solving design problems of a complexity similar to what the students will see after graduation.

We have selected the medical school clinical model to implement the practice element of the final two years of the software engineering degree. The reason we have chosen the medical school model is that it allows us to couple fundamental learning with practice-based learning. Our clinical rotations are centered on the major activities a software engineer performs. Each rotation has a foundation component and a practice component. As an example, software requirements elicitation methods, representations, and validation techniques would be coupled with the development of requirements for a project. There are four attributes of our proposed curriculum:

1. The program is based on two years of basic computing education followed by the "clinical" practice, similar to most medical schools.
2. The practice elements are on-going projects that students participate in. Similar to patients coming and going in a hospital and offering students a view of different diseases and their treatment, our projects offer students a set of views of different projects at varying stages of development. But like medical school, the rotation has a basic component of practice that is being emphasized. That is, neurology and obstetrics aren't mixed into the same rotation in medical schools, and requirements and coding aren't mixed into the same rotation in our program.
3. All students in the program don't follow the same lock step sequence of rotations. That allows projects to be on-going with students participating in particular activities as a function of the project and their rotation. Similarly, students aren't necessarily assigned to one project throughout their practice.
4. The division of foundations versus practice in the rotations was made based upon our analysis of the knowledge elements of software engineering and a division of those elements into skill and knowledge categories. Those divisions allowed us to allocate learning objectives between the foundation and practice parts of each rotation. Each rotation therefore has a set of foundations, more appropriately taught in the classroom, and skills more appropriately learned through practice.

4. The Software Factory

The previous section dealt with the concepts of rotation in a generic sense. We now focus on our implementation of rotations for software engineering students, which we call the *Software Factory*. Unlike other “practice-oriented” models such as real-world

labs, clinics and studios where the classroom instruction is separate from the practicum, the Software Factory integrates them in a synergistic fashion. Rather than register for classes individually, students simply register for the Software Factory. Within the Software Factory, all software engineering academic and practical experience is provided for the students.

Students in the software engineering curriculum take 9 hours of the Software Factory each semester of their junior and senior years. We divide each semester into two parts giving us 8 rotation slots for organization of classroom and laboratory exercises. Figure 1 depicts an idealized rotation through the Software Factory. The upper line of slots represents the academic or classroom-type instruction provided to students. It is here that the fundamental concepts of the rotation phase are discussed and students receive the "just-in-time" education to support the practicum rotation they are involved in.

The bottom line of the figure depicts the student's progress through the Software Factory in terms of the practicum topics. These practicums are not traditional classroom projects, but rather are phases of a real project for a real customer. The students work on these real-world projects applying the skills and knowledge they are simultaneously learning in their classroom rotations to accomplish the goals of the project.

Fall Semester		**Spring Semester**	
Boot Camp	Requirements Engineering	Design I	Design II
Construction	Requirements		Design I
Fall Semester		**Spring Semester**	
Reengineering/ Maintenance	Project Management	Quality Software Development	Elective
Design II	Maintenance	SQA/Testing	Project Management

Figure 1 - Software Factory Idealized Rotation

4.1 Classroom Rotations

There are 8 class rotations in the Software Factory. These rotations include: boot camp, requirements engineering, design I, design II, project planning, reengineering and maintenance, and quality software development. Recall that in the first two years of the program, the principles of "programming in the small" were covered with the students. The Software Factory on the other hand, focuses on the problems and techniques associated with "programming in the large." The order which students take class rotations is not prescribed except that boot camp must be the first class taken. Boot camp serves two primary purposes. First, it is a rotation slot where miscellaneous subjects that don't require a full rotation can be taught to the students. The second and most important purpose of boot camp is to serve as an "indoctrination" into the Software Factory and its processes.

The remaining classes are fairly recognizable as major topic areas in large-scale software development. It is important to note that although the topic names have a distinctive "waterfall" feel to them, the Software Factory is not constrained to any one

particular life-cycle. Rather, these topics serve as descriptive labels for groupings of major concepts that can be applied to discrete parts of a Software Factory project. Again, borrowing from education in law, architecture, and medicine, classes will couple teaching with analysis of project case studies. The case studies will be drawn from the repository of Software Factory projects. The analysis will consist of identification of process, best and worst practices, and pitfalls over the course of that project's lifecycle within a particular rotation. We believe this approach will help students to begin making connections between theory and practice, preparing them for the practicum rotations.

4.2 Practicum Rotations

Unlike the 8 classroom rotations, which have a set length and schedule of activities, the practicum slots are more flexible. Some practicum slots may be adjusted longer or shorter based upon the specific requirements of the project that the practicum supports. Students in the practicum work on a specific project selected from among those currently being developed by the Software Factory. Like the classroom rotation slots, practicums may be executed in any order—with the exception that the construction rotation is always conducted first (in conjunction with boot camp). The construction practicum might appear redundant with the course work in the first two years, but remember that the Software Factory construction practicum deals with construction of software in large-scale development efforts. Students use configuration management tools to manage code changes, perform group code reviews and accomplish other tasks indicative of the complexities of developing code modules as part of a team, rather than as an individual.

In the project management practicum students may have leadership responsibilities either directly in support of a specific project, or at a higher level in terms of management of the Software Factory itself. Likewise, the software quality assurance practicum ranges from duties directly in support of testing for specific projects to those of quality initiatives at the Software Factory level.

4.3 Extra-Curricular Responsibilities

There are many extra duties software engineers fulfill outside their regular jobs. Many of these duties expose students to more depth than they would receive in a regular rotation. In the Software Factory, students are members of various boards and groups that manage specific activities in the Software Factory. These include the: Configuration Control Board (CCB), Software Engineering Process Group (SEPG) and the Quality Management Board (QMB). Senior students may also act as mentors for entering Software Factory students.

4.4 Business Model

The business model for the Software Factory is fairly simple. Projects and students flow into the Software Factory, while completed work artifacts (documentation or actual software) flow out. The Software Factory management obtains projects from customers, such as the College of Computing, the Georgia Tech community (business and academic), and from industry participants. Projects will be assessed for difficulty and duration before being released to the project groups. Students in the project management rotation scope the project and create a preliminary project plan. The project is then assigned to a student project manager for execution during the next rotation. While the type of rotation the project will support during the next rotation is driven by the life cycle

chosen by the project managers, it generally supports the requirements rotation on its initial entry into the Software Factory.

As the project continues through the Software Factory, student managers in the project management phase assign the project to the next rotation that it will support. The Software Factory is designed such that each phase of the project development does not have to fit into one rotation cycle. Project activities can be just part of a rotation or can span multiple rotations. Different activities can occur in parallel or lockstep as needed. The point is that the Software Factory allows students to make realistic decisions in light of realistic constraints.

Obviously, there is some artificiality in the Software Factory driven by educational and assessment goals, but the overall desire is to emulate a realistic software engineering environment. The challenge is to balance the desire for realism with the need for scaffolding and mentoring the students to ensure they not only accomplish project milestones, but they accomplish them in the right way. The next paragraphs will discuss some of the practical requirements to support day-to-day operations of the Software Factory and some issues that need to be considered.

5. Implementing the Software Factory

The Software Factory should have an actual physical identity and model to support the projects correctly, as opposed to a virtual simulation that the students would operate in through existing labs and meeting rooms. Like the teaching hospitals of the medical school model, the Software Factory will provide the students with working facilities, such as testing rooms and a software engineering library, teaching and support personnel, and real "patients". We would like to implement an actual working environment that accepts and delivers real software projects where a student's actions can have real consequences besides their grade. Such an environment will provide the necessary immersion for the students to learn how real projects are run and will also reinforce a sense of professionalism in their approach to the work. The next sections discuss personnel and organizational issues involved in implementing the Software Factory.

5.1 Personnel

In order to ensure proper Software Factory operations for both teaching and day-to-day operations, it will be necessary to have a mixture of academic instructors, technical instructors, project group managers, and support personnel. To supplement the academic component with knowledge of current industry best practices, the Software Factory will hire technical instructors who have worked in industry as software developers and can serve as consultants for project groups. We recognize that skilled software engineers from industry will be in high demand and therefore hard to obtain. One possibility will be to have these technical instructors visit on a part-time basis as part of a working partnership with an external company. In return, these companies will have first contact with pending software engineering degree graduates. Lastly, the Software Factory will require a general full-time manager and possibly additional service personnel for day-to-day operations and to support the variety of tools and computing equipment required by the projects.

5.2 Organization

There are two basic organization structures in the Software Factory: instructional and management. The instructional side handles all issues related to the education and training of the students. Instructors will be responsible for teaching the classroom rotations and organizing the rotation schedules, and acting as technical consultants for project groups. There will be a number of teaching assistants and senior students who will also help with mentoring, meeting with project groups and with their project managers. The management side handles all issues related to the maintenance of the technical infrastructure, management of project groups, the coordination of instructional and student resources to the projects, and community outreach, such as customer contact and service, industry recruiting and hiring, and project searches. While instructors should meet regularly with the project groups to check their progress and to offer advice, we expect that the full-time manager and student managers will deal with most of the management issues. We discuss some of the logistical difficulties in a later section.

6. Difficulties

6.1 Unfamiliar Territory

We recognize that there will be some problems associated with our relative unfamiliarity with implementing this educational model. There may be problems convincing a department that this model is feasible. The startup costs are more expensive than a typical curriculum. Aside from the costs associated with constructing new coursework, we will have to consider the start-up costs associated with additional equipment, materials, infrastructure, and personnel dedicated to the running of the Software Factory. There will also be problems associated with the maturity of the Software Factory. It may take a few years to build up a decent repository of cases for the classroom rotations, sufficient projects in different stages for student projects, as well as sufficient experience in handling the operations of the Software Factory. For example, we will not have the experienced senior students to manage the first sets of projects. Thus, Software Factory personnel and graduate teaching assistants are likely to have additional burdens and tasks the first few years. However, we believe these problems will be mitigated with time.

6.2 Managing the Software Factory

The Software Factory will be very difficult to manage for several reasons. The number of project groups, rotations, types and lengths of projects, customer relations, and facility management will be a logistical nightmare. Ideally, the flexibility of the Software Factory and the rotation model will allow us to create any size group with an appropriate management structure to handle any project. However, too many projects will create potential pipelining problems with some of the time-critical rotations such as development and testing. Projects may finish in the middle of a rotation or have to be stopped due to insufficient students in a particular rotation. Projects that fail to meet the expectations of a project plan can create scheduling problems for Software Factory managers and instructors. In a worst case situation, the instructor and Software Factory management will have to determine why the project failed to meet deadlines and determine whether that group has to be sent through the rotation again, delaying the

graduation of the group members and the completion of the project. We will probably have to learn how to handle these issues, both on a case-by-case basis and through policies that will evolve with our understanding of Software Factory logistics. We can take advantage of much experience in smaller scale projects that are already common for senior computer science students. Scaling this experience will require a full-time manager for the Software Factory.

Another obstacle will be managing exceptions within the Software Factory. The Software Factory must be able to handle students who become ill or are absent for a time. These students need to be given opportunity to make up work, but their teammates must not be penalized. The Software Factory must also allow for co-op students or others who are not at school for certain semesters. Rotations must be offered frequently enough so that these students will still be able to graduate in a timely manner. The current curriculum revolves around 2 semesters a year. We should also consider possibilities for offering shortened summer semester rotations.

6.3 Assessment

A primary difficulty in successfully implementing the rotation model is assessment of the students. As mentioned previously, we wish to provide an environment where the students produce real products for real customers. However, we also wish to provide the students with enough scaffolding so that they accomplish this in the right manner and so that they have the flexibility to fail and learn from their failures. Thus, the emphasis needs to be both on the product they are producing, and the process with which they produce that product. This may require careful and frequent observations on the part of the faculty and students managing the Software Factory. There is also danger of subjectivity in the grading. For example, project managers are being assessed for their performance in a project management rotation while their team members are being assessed for their performance in their rotation phase. If the project fails, does the responsibility fall on the manager or on the team?

We have not yet decided on one particular assessment method for the Software Factory. Many factors could be used to grade students: feedback from fellow teammates; reflections of the students on their work and process; peer reviews of the product; weekly status reports, design drafts, or meeting summaries; and feedback from teaching assistants or projects leaders.

7. Conclusion

We believe that software engineering education is lacking a vital practice component in providing students both the skills and knowledge they need to successfully apply their degrees in industry following graduation. We have proposed a model for a curriculum of a software engineering bachelor's degree that addresses this concern. Our rotation model will provide a realistic setting where student software engineers can try and hone their software engineering knowledge.

Georgia Tech is planning on offering a bachelor of science in software engineering starting Fall 2001. Our goal on this project was to develop a software engineering curriculum that can be integrated into our existing computer science curriculum, yet offer software engineering students the practice based component that we feel is central to a software engineering education. We are actively exploring implementing our proposed curriculum and addressing the issues we have raised in this paper.

8. Acknowledgements

Our thanks to the other graduate students who participated in the creation of this curriculum: Ivan Brusic, Peter Chan, Deborah Esslinger, Jason Greenidge, Brian McNamara, Terry Shikano, Lex Spoon, and Dinggang Xu.

9. References

[1] IS 95, *The result of the joint task force of ACM/AIS/ICIS/DPMA*. Paper presented at the Decision Sciences Institute, Honolulu, HI, 1994.

[2] Bagert, et. al., *Guidelines for Software Engineering Education, Version 1.0* . Pittsburgh: Software Engineering Institute, 1999.

[3] Barnes, et. al., "Draft Software Engineering Accreditation Criteria", *Computer,* IEEE Computer Society Press, April 1998, 73-75.

[4] J. R. Dixon, "New goals for engineering education", *Mechanical Engineering,, 113,* 1991, 56-62.

[5] W. Newstetter, "Of green monkeys and failed affordances: A case study of a mechanical engineering design course", *Research in Engineering Design*, Oct 1998, 118-128.

[6] Richter, H. A., Waters, R., McCracken, W.M, Hsi, I. (2000). *Building on the Rock (under review).* Paper presented at the Conference on Computer Science Education.

Appendix A

Advanced Software Engineering	Goal Level	Practice (%)
Group Process		
Organizational roles / structure	3	50
Team building	3	95
group communication	3	95
group dynamics	3	50
Maintenance		
Reuse		
Unplanned		
Evolution of a system	4	90
Reusing components in new system (new design, old components if possible)	4	90
Planned		
Design/implement for maintainability	3	60
Sharing generic components across projects	3	60
Reverse Engineering		
Why? (maintain/evolve a system, discover Valuable business policies/algorithms)	1	0
Static analysis (with and without automated Assistance)	1	0
Dynamic analysis	1	0
Configuration Management		
CM for product and process	Diffuse	100
Process		
Models/lifecycle	3	50
Process modeling and languages	1	0
Project Management		
Project Design and Planning		
Project taxonomies and objectives	2	20
Project Lifecycle Planning		
Decomposing the project into distinct phases and achievable processes	4	80
Evolutionary and staged delivery models of planning	2	10
Rapid Development and Prototyping		
core industry process for development	4	95
variations: what if another model is required? More time or resources are available?	1	0
Project postmortem	Diffuse	100
Resource Allocation		
Effort Analysis and Estimation		
basic business economics – return on investment, salary costs, equipment costs, etc.	2	50
project estimation	3	50

Advanced Software Engineering	Goal Level	Practice (%)
Resource allocation	3	50
Risk Management		
Risk Taxonomies and Identification		
types of risks	2	0
characterizing the project's risk tendencies	3	50
Risk planning	3	50
Project scheduling	3	50
Project tracking & control	3	50
Requirements		
Elicitation	4	60
Representation	4	60
Evaluation	4	60
Specification	4	60
Design		
Software architecture	4	60
real time systems	2	0
Information systems	2	0
Distributed systems	3	60
Refinement	4	95
Product & Process Quality		
TQM		
Shewart cycle	2	0
SPC and seven basic tools	2	0
Software Process Improvement		
CMM	3	60
personal process management	4	80
ISO 9000, SPICE and other models	1	0
safety – hazard analysis	2	0
V&V	4	60
Metrics	3	60
HCI		
Interaction patterns	2	0
Usability principles (learnability, flexibility, etc.)	2	0
Usability testing	3	30
Interface prototyping	3	70
Design rules and standards	1	0