The Education of a Software Engineer

Mehdi Jazayeri
Technical University of Vienna
mehdi.jazayeri@tuwien.ac.at

Abstract

A successful software engineer must possess a wide range of skills and talents. Project managers know how difficult it is to find, motivate, and retain such people. Educators face a complementary, and perhaps more challenging, problem: how to prepare such engineers. The challenge of what to teach software engineers evolves over time as technologies, applications, and requirements change. As software technology has rapidly spread through every aspect of modern societies, the challenge of educating software engineers has taken on new form and become more complex and urgent. In this talk, I present the broad outline of an educational program for a complete software engineer. A new curriculum for computer science has been developed based on these ideas and will start in October 2004 at the University of Lugano in Switzerland.

1. Introduction

Over the years, the teaching of software engineering has changed only slightly and most textbooks of software engineering follow rather traditional and similar lines. This might give the impression that there is general consensus on what must be taught to software engineers. On the other hand, listening to discussions between academics and practitioners at conferences reveals deep disagreements. Many practitioners believe that universities are not doing a good job and many academics argue that industry does not use the latest, best technology.

There are deep, underlying, reasons why universities have difficulties in educating software engineers. A software engineer must combine formal knowledge, good judgment and taste, experience, and ability to interact with and understand the needs of clients. It is not easy to teach all of this, certainly not in one or two courses on software engineering!

The typical courses on software engineering concentrate on the phases of the development process: requirements analysis, specification, design, implementation, and testing. In recent years, some of these issues have been enhanced because of new research results. For example, requirements is now treated much more systematically, often using UML as a standard notation. Software architecture has become a standard topic, providing a bridge from requirements to design. Still, most of the emphasis is on these forward engineering development steps. Some courses also cover management aspects, but mostly in a theoretical sense. These treatments leave out the entire “experience” aspect: it is one thing to read about how to design a module and quite a different thing to design a specific module that is supposed to meet specific requirements and fit in with modules designed by others. To provide some level of experience with the techniques that are taught in textbooks, many courses include, or are complemented by, a project component in which the students, usually in small teams, develop a medium-sized software application. While the aim of the project is to show the student what the “real world” of development is like, and often these projects are the most time-consuming projects that students undertake in their studies, by necessity the project must be constrained to ensure that it can be completed within the semester with a reasonable amount of work. Sometimes the instructor specifies the requirements, hiding the most difficult aspect of the real world, in which the requirements are never really known. Usually the infrastructure and the development environment are pre-determined and are ones that the students are familiar with, the schedule is fixed and the project has already been trimmed to an appropriate size by the instructor, and there are no compatibility or legacy requirements. None of these constraints reflect the real world of software projects but they are necessary for practical reasons to respect the academic calendar in which semesters come to a quiescent end, with no possibility of lawsuits or contract disputes!

The solution to these challenges is to design a whole new curriculum of computer science that integrates the different topics that we teach and addresses the new realities in the application world. Because of the importance of software, its engineering, and how we teach it, is a core component of such a curriculum. In this paper, I will attempt to outline the challenges of teaching software engineering today, the emerging requirements that software engineers must satisfy, and propose a modern curriculum to address these challenges. A curriculum of this type has been designed in the last two years and will start for the first class of students in October 2004 at the University of Lugano.

2. Traditional challenges of Teaching Software Engineering

Teaching software engineering has never been easy and no consensus has emerged from the many debates about how best to do it. At the base of the problem lies the fact that the complexity of software engineering comes from the complexity of problems and it is impossible to construct complexity in a classroom setting. Indeed, the purpose of classroom teaching is to peel enough complexity away that the problems become doable by students. In software engineering, unfortunately, if you peel away complexity, you are left with unrealistic (sometimes called toy) problems.

Another difficulty of teaching software engineering is that it is a multi-faceted discipline. As a result, there are many tradeoffs that an instructor must make, thus limiting the experience of the student. Some of the common tradeoffs are:

-Practice versus theory. How much should we teach about current state of the practice and how much about an idealized approach that our theories cover? The theoretical approach emphasizes the importance of formal specifications, program verification, and in general a disciplined and systematic approach to software development. In the practical approach, one emphasizes the difficulties that arise in the real world, despite taking a systematic approach to software development. These problems range from unreasonable customers who can't make up their minds to difficult colleagues who refuse to change their interface to accommodate new requirements to incompatible versions of the version control system. This tradeoff is a manifestation of what Fred Brooks [1] has described as essential versus accidental complexity of software. The theoretical approach deals with the essential complexity while the practice-oriented approach deals with the accidental complexity. The usual solution to this tradeoff is to combine a lecture course, dealing with the theory, with a laboratory course, in which the students face practical issues. With the caveat that one cannot re-create the real world in a classroom, close approximations to the real world are possible.

-Development versus management. From the birth of software engineering, some have viewed the problems as being primarily managerial and others as primarily development-oriented. One could teach about how to form teams and establish proper communication channels or one could teach about module design. One could teach about making the quality assurance team independent from the development team or one could teach about testing techniques. In reality, the differences are not so sharp and the most important problems cross the boundaries of development and management. For example, as Parnas [6] has pointed out, the essential concept of modularity that guides module design can be used as the basis for work assignment in a project. An architectural approach to development supports better management practices.

-Product versus process. Should we teach about the software object and its constituents or about how we construct the software object? In the former approach we emphasize the programming and other languages and in the latter we emphasize at what step we should use those languages. In the product-based approach we emphasize more design issues and in the process-based approach we emphasize the problems that occur in the process. This tradeoff has always been the focus of software quality improvement approaches. Capability Maturity Model (CMM) is the well-known assessment approach that measures the software production quality of an organization solely based on process-related issues.

-Formal versus empirical. This tradeoff is between learning by studying versus learning by doing. The two schools of thought view software engineering as a mathematical science or as an empirical science. Empirical software engineering emphasizes experiments and statistics to characterize the results of those experiments. A laboratory approach to some degree supports the empirical approach

Of course, most textbooks and most courses try to cover all of the above aspects, making more or less conscious choices about the tradeoffs. If we were sure what the graduate of the course would end up doing, it would be easier to decide on the tradeoffs but we usually do not know that. A textbook, in particular, tries to address a general audience and for software engineering the general audience is not very homogeneous. In any case, the necessity of making choices among these tradeoffs has made the teaching of software engineering a challenging task.

3. New realities

The traditional challenges of software engineering have been exacerbated in recent years by the growing importance of software and by new technological developments. In this section, I discuss what I consider to be the trends with the most significant impact on the teaching of software engineering.

-Distribution. The change of computing platform from mainframe computing to distributed computing requires a fundamental reconsideration in the way we view the structure of software and the basic notion of modularity. Historically, software engineering practice, and software engineering textbooks too, have dealt with centralized (mainframe) software systems. The often-maligned metric of lines of code to measure the complexity of software or the productivity of programmers is an indication of this mainframe bias. Today's world, however, is distributed by default. This means that we must begin with distribution as a starting point rather than as a special case of software engineering. To what degree must we teach about communication, synchronization, caching, security, fault-tolerance, and other such concepts that are traditionally the domain of distributed systems? In fact, no software engineer can ignore these issues and they are complex enough that they must be engineered if they are to be included in any system. This argues for a closer coupling of software engineering and distributed systems. But if we already had too much to teach in a software engineering course, how are we going to include the new distribution considerations?

Another aspect of distribution involves the people and processes in software engineering. Increasingly, software is being developed by teams of engineers that are geographically, and often also organizationally, distributed. While this does not change the formal or theoretical aspects of software engineering, it does have a fundamental impact on the processes, tools, and practices that can and should be used. There are interesting ways that traditional formal concepts are affected by such practical realities. For example, the practice of outsourcing makes some traditional concepts such as module decomposition and program specification more important than ever before.

-Pervasive computing. Pervasive computing seems to be the technological trend of distribution extended to its extreme. There are several different views of pervasive computing but the common one refers to the availability of unlimited computing and communication elements in the environment, where every object can communicate with other objects. Pervasive computing requires us once again to question our assumptions about the structure of software. The most important difference is the introduction of dynamicity. The software must be able to deal with computing elements that enter and leave the environment at arbitrary times. Software services and applications must be created dynamically out of these computing elements. This dynamic world is at odds with traditional software engineering in which we try to fix (bind) as much as we can as early in the process as possible. Traditional software engineering favors static decisions and pervasive computing forces most decisions to be made dynamically.

-The Internet. Of course, the Internet has changed everything and software engineering is no exception. The Internet has had several different effects on software engineering. First, it is used as an execution platform. Second, it is used as a development platform. Third, it is used as a delivery vehicle for software. Each of these engenders its own version of software engineering issues. For example, using the Internet as an execution platform can be based on various Internet protocols and security mechanisms and infrastructure. We are beginning to see book titles such as Software Engineering for the Internet. There are even degrees offered on Web Informatics.

A more direct impact of the Internet on education in general, not only in software engineering education, is the availability of an unlimited reservoir of information. For example, the open courseware library offered by MIT on the Internet (ocw.mit.edu) is a wonderful source of material for instructors and students alike. It is not clear, however, how best to exploit this information in a traditional course setting.

-Proliferation of software tools and environments. Practitioners and educators alike have always recognized the importance of tools and environments to support software. The state of practice, however, has changed considerably over time. At some point in time, tools were used to enforce certain methodologies or company policies. The adoption of such tools by companies is a major decision with large impact on the processes and future decisions of the company. The assumption is that the experience of the company's engineers in the use of the tools is so valuable that the tools must be used over a long period of time. Adopting a different toolset or environment discards the hard-earned employees' experience. Furthermore, a company's software engineering processes build around the set of tools being used. Changing tools requires changing processes, which is also an expensive undertaking. As a result, the state of tool adoption in industry is rather static and conservative. On the other hand, over the last decade, there have been tremendous developments in the area of software tools, spurred in part by the very active research on software environments in the 1970s, in part by the open source movement, and in part by the existence of the Internet platform for execution and delivery. The pace of technology development and supporting tools has picked up so considerably that the conservative approach of earlier times is no longer viable. A software engineer must now be able to pick up a new tool and an associated process quickly, regardless of his or her vested experience in a previous tool. Internet-time has certainly affected tools and processes and requires agility on the part of the software engineer. On the one hand, this implies that the choice of, and emphasis on, tools in a software engineering class is not as important because almost certainly the students will use very different tools in their profession. On the other hand, it implies that the student must gain the skills to be able to switch among tools. Where and how is this skill to be acquired?

-Software evolution. It is now generally accepted that the initial focus of software engineering on development ignored the importance of software evolution. We now acknowledge that we have problems with legacy software that must be maintained by software engineers. There is significant research currently going on in the area of software evolution, from theories to tools to processes. It is more likely that a software engineer will be employed in software evolution than new software development. Should this new reality shift the emphasis we place in software engineering courses from software development to software evolution? More likely, we have to seriously consider evolution alongside development. But we face the same problems of creating a realistic evolution experience in the classroom. What makes evolution difficult is the size and complexity of the software, company organizational issues, and the necessity of parallel development. Recreating these issues in the classroom makes the class unmanageable and restricting them to make the class manageable defeats the learning objectives.

-Software quality. Complaints about software quality are commonplace. Most of us acknowledge the shortcomings in today's software but few of us do anything about it. As software pervades society's infrastructures and runs most of its services, software quality cannot be ignored. With emerging pervasive computing applications and services, software quality becomes even more challenging. Software engineers must have tools and techniques to build high-quality software. Where do they get these tools and techniques? Most software engineering courses and textbooks go little beyond testing techniques in this area. The problem here is a lack of concrete techniques. The best that can be done today is to impart a sensitivity to quality issues. Software engineers must learn that "time-to-market" is not the only measure of project success. They must learn tradeoffs that take into account quality factors as well as more concrete factors. This area needs much more research but educational needs are rather acute and cannot wait for research results. The only way I know to emphasize this subject is to teach the responsibilities of a professional software engineer. This has to be a critical part of a software engineer's education. There is not a wealth of material on this subject but a good starting point is [2].

-Computing platform. With a distributed computing environment providing the hardware platform, middleware provides the software platform, something analogous to operating systems of the past in centralized systems. Most middleware systems offer similar facilities but they also have significant differences. Software engineering on top of different middleware systems could be based on different approaches and techniques. Should a middleware platform be part of the study of a software engineer? While we would like to believe that the concepts and principles are independent of the actual middleware, a skilled software engineer needs detailed knowledge of the middleware principles and the gap between general concepts and concrete middleware practices is growing rather large. The choice of which middleware platform to use in classroom teaching involves similar considerations as the choice of what programming language to use. Should we choose a platform for its teaching value or for its current popularity? A disturbing trend is the mixing of marketing and technical considerations. The argument that one platform is better because it has more users is not a valid argument for educational choices. Regardless of how fast technology develops, we in universities must prepare engineers who will cope with the technologies of at least the coming decade.

-Interdisciplinary informatics. Pervasive computing refers to a branch of (distributed) computing that considers abundant computing power embedded in everyday environments. The field lies at the intersection of embedded and distributed systems. We can, however, interpret "pervasive" computing in a more general sense, in the sense that computing is now pervasive in all aspects of society, ranging from business, to government, to education and science. No profession can function or advance without computing. New drug discoveries, new material inventions, new business products and processes, are all based on heavy use of computing and software. This means that software is being developed to address the needs of many diverse disciplines. Indeed, we once taught informatics as if every computer scientist was going to work with other computer scientists. But today most computer scientists and software engineers will go to work with non-computer scientists working in different, sometimes novel, application areas. New application areas of computer science are emerging. While some of them have names—such as bioinformatics—many software engineers find themselves working in emerging interdisciplinary informatics settings that do not have names yet. An interdisciplinary software engineer must be conversant in other disciplines, perhaps able to work with different disciplines. This ability relies on strong abstraction and modeling skills, two skills that are essential in software engineering and are even more so for interdisciplinary informatics.

4. Non-technical skills

So far, I have discussed only technical skills required by a software engineer but there are some non-technical skills that are also essential to the success of a software engineer. The two most important such skills are communication and the ability to work in a team.

Communication. A lot of the time of a software engineer is spent communicating with others: with clients, peers, managers, suppliers, and others. Communication is indeed the basis of requirements engineering. Documentation is the most concrete example of communication. Sadly, managers at companies often complain that engineers, even those trained at best universities, are deficient in both written and oral communication skills. Communication is more than using a language according to correct grammar, in writing reports or in making speeches. What is difficult is to choose the right level of abstraction depending on the subject of discussion and the communication partner. Modeling skills are also an important requirement for successful communication. Indeed, since requirements-related problems are the most costly problems during software development, and communication problems are a major source of such requirements problems, we must invest a lot more in educating software engineers in the area of communication. There are probably different kinds of communication techniques required to interact with peer software engineers, with those in other disciplines, with managers, with clients, and so on. In all cases, however, listening is an important part of communication.

Ability to work in a team. All software engineering projects are executed by teams. And it is well-known that effort spent on a project goes up proportionally to the square of the number of people involved in the project. This is believed to be due to the overhead of increased communication. Certainly, good communication skills can help one be a better team player, and reduce the communication overhead, but good communication skills is not enough. Working in a team requires making room for others, making compromises, asserting oneself when necessary and accepting others' judgments when needed. We do little in university education in general, and in software engineering courses, in particular, to teach teamwork. In fact, we usually emphasize individual skills and ignore the role of the individual as a team member. Indeed, the role of the individual is sometimes even glorified as the one who comes through at the last minute to save a project. Clearly, organizations that rely on such heroics are not engineering-oriented.

We must of course recognize that one's attitude towards teamwork, working with others, accepting authority, and other such matters is heavily influenced by one's cultural upbringing. We therefore cannot expect principles that apply throughout the global landscape. What we have to keep in mind is that software is developed by teams and a software engineer is only one of a number of people working on that team. Therefore, the engineer must learn not only how to produce his or her modules, and how his or her modules must fit with modules produced by others, but also how he or she must fit within the team.

There are technical solutions such as module decomposition and tools such as configuration management systems that help support better workings of a team. Here, I am concentrating on the non-technical skills required. These skills are even more drastically required—because the problems of teamwork are magnified—when we have distributed teams of engineers. Software engineers rely on social and casual contact for important communication—a luxury that is not available to geographically distributed teams.

A final important quality of a software engineer is experience and good judgment. Clearly, experience only comes with time. Good judgment, it is said, comes from learning from making lots of mistakes. The role of engineering education is to create an environment in which mistakes can be made that will help in developing good judgment.

5. Ingredients of a curriculum

Considering the traditional and emerging challenges facing a software engineer outlined so far, it is unlikely that one, or even a few, courses on software engineering can prepare a software engineer for the real world of software development. What is necessary is an integrated curriculum that tries to cover the many facets of software engineering or, indeed, a software-engineering focused curriculum of computer science. In this sense, I interpret software engineering in a broad sense, almost equating it to computer science. In fact, if we consider informatics as the study of concepts and methods that enable the creation and manipulation of software, software engineering is at the core of computer science. All of the techniques of software engineering: problem-solving, problem definition and specification, planning, scheduling, verification, documentation, and so on are ingredients of any other fields of computer science such as databases, compilers, and graphics. Therefore, I consider here a curriculum for software engineering or a curriculum for informatics based on software engineering principles.

In summary, the challenge of designing a curriculum for informatics today is to find a way to combine formal with practical learning, technical with non-technical skills, and informatics with interdisciplinary knowledge. To do this, we need to, as much as possible, create a real-world environment at the university. The purpose is to enable the learning of non-technical skills in a formal way. This environment can be created in the context of carefully designed projects. These projects should be integrative and comprehensive, rather than associated with a single course. On the basis of these projects, we can create a project-focused curriculum. I suggest that students should spend about half of their time in classroom learning and the other half on projects. Each semester is structured with specific courses and one semester-long project. The goals of the project, which vary in degree from semester to semester, are:

- to show the application of classroom theory to practice
- to integrate the material from various subjects
- to teach the proper use of tools
- to show the relationship between accidental and essential complexity
- to enhance teamworking ability, including communication skills

Clearly, due to the key role projects play in the curriculum, they must be designed and administered with care. For example, consider the teaching of programming. Typically, we teach the concepts of programming languages along with some syntax of the language, and leave the students to struggle with compilers, interpreters, and other tools. Fortunately, today's tools are much better than we had ten years ago and they do not give as many incomprehensible error messages as they once did. On the other hand, a compiler is not the only tool one needs for programming. Other useful tools are: editing and testing, configuration management, defect tracking, help system, documentation generator, and so on. A systematically-guided project should teach the student about the whole programming environment rather than just a collection of tools. Such an approach also helps the student see the bigger picture of software development in the context of a multi-person project, rather than just a programming exercise. The semester-long projects can be designed in increasing level of sophistication, initially involving individual work and gradually leading to small team projects, larger team projects, and larger interdisciplinary team projects. The project environment can also be used for holding formal classes on project management, covering such topics as project estimation, scheduling and reporting. Special seminars on communication techniques and teamwork can draw on specific project experiences.

6. A planned curriculum

In the last two years, I have been involved in designing a curriculum for a new bachelor's degree in informatics at the University of Lugano (official name in Italian: Università Svizzera Italiana) in Lugano Switzerland (www.unisi.ch/en/informatica). The bachelor's degree, as mandated by the Bologna Convention of the European Union, requires three years to complete. The Bologna Convention envisions that all European universities will have a 3-year bachelor's degree followed by a 2-year master's degree. The uniform length of the bachelor's degree program is expected to facilitate "mobility" among universities, enabling students to pursue their master's degree in a different university than the one in which they earn their bachelor's degree.

This curriculum in Lugano is based on the ideas I have presented in this paper. In this section, I discuss some of its most interesting aspects.

Overall structure.

The curriculum is structured around five different areas that are essential for an interdisciplinary education in informatics.

Theory. Any scientific discipline has its theoretical underpinnings that are essential for the study and understanding of the subject. Regardless of how we emphasize practical issues and the real world, a theory is necessary for the identification, specification, and analysis of problems. No amount of communication skill can make up for lack of theoretical knowledge. There is debate in computer science and software engineering about what this theory is and how much of it is necessary for students. We have decided on discrete mathematics, logic, analysis, statistics, along with theoretical computer science.

Technology. One of the problems faced by students and instructors of computer science is the rapid pace of technology. Clearly, theory should be taught independently of current technology. It should cover principles that will last years—perhaps for ever—and survive several generations of technologies. On the other hand, technology is important in informatics and for software engineers because it progresses so fast and its advances make qualitative differences in what is possible. Also from a practical point of view, graduates should be familiar with (some of the) current technology, whether they are going to work in the commercial world or take part in research projects. In software engineering, technology covers such fields as programming languages, operating systems, middleware, and programming environments. In this part of the curriculum, students encounter several different such technologies.

System approach. Software is almost always part of a larger computer system. And a computer system is also almost always part of a larger system itself. It is therefore important for a software engineer to be able to take a system view of problems and solutions. The building of large systems and predicting their behavior and their impact once they are deployed is notoriously difficult. It is certainly difficult to teach system-level thinking in a single course whose emphasis is on a particular topic. For example, an algorithms course must concentrate on algorithms and a database curse must concentrate on models and algorithms for databases. Putting together a system that includes a network of databases, web servers, and client machines usually falls outside the purview of any single class. Thus it is not easy for a student to get the "big picture." The curriculum tries to impart system-level thinking through the use of semester-long projects. For example, one can start with an existing system (perhaps built in a previous semester), analyze its behavior and performance, and plan and implement extensions and improvements to the system, followed by further analysis of the system. Another aspect of systems thinking is to consider not only the technical aspects of building the system but also the impact of the introduction of the system on the behavior of the people and processes in the organization. Also this aspect requires an interdisciplinary approach.

Interdisciplinary applications. Anticipating that the graduate's career will involve working in different application areas, the curriculum attempts to familiarize the student with several different application areas. This is done in two ways. One is to assign projects that deal with different application areas, ranging from life sciences to economics and business. Unlike typical projects in computer science courses in which the emphasis is on how to apply learned computer skills, these projects emphasize solutions that benefit the application area. The plan is to form teams that include software engineers but also members from other faculties in the targeted application area. The second way in which an interdisciplinary approach is pursued is to teach the models and modeling techniques of several other disciplines. Life science models and economic models are covered as part of the curriculum with this goal in mind. The reasoning is that if software engineers are familiar with models used by specialists in application areas, they are better able to understand the problems of the application area and to communicate with their clients.

Communication and teamwork. This part of the curriculum emphasizes so-called "soft" or behaqvioral skills. As argued earlier, these skills are especially important for software engineers. These skills are taught with the help of semester-long projects, complemented with focused seminars on theories and techniques. The importance of communication can be seen in requirements engineering, especially in interdisciplinary projects. But students will see first-hand the need for communication, and its difficulties, in team projects.

To enhance the emphasis on teamwork, the building that is to house the computer science department has been designed to include "teamwork areas." In fact, the requirements for the architect called for areas that support teams of students working together. Each floor, rather than containing large laboratories, includes modular structures that can be set up to accommodate different team sizes. Each team area has its own infrastructure support for the team.

Teaching methodology.

As can be seen from the discussion in this section, the curriculum makes heavy use of, and is reliant on, the educational value of projects. Such project-based teaching has been gaining interest in the last few years in different areas. Some disciplines, such as architecture, have a long tradition of project-based education. In other disciplines, its adoption has followed advances in educational theory and technology. There are various reasons cited for the adoption of a project-based approach. One is that it engages the student and therefore increases motivation. Another is that in certain fields (more like crafts) learning by doing is the most effective. The former reason is cited by MIT as its motivation to recently convert its introductory physics course to be project-based. The latter reason is why many architecture schools adopt a project-based curriculum. For informatics, the situation is different. As I have tried to argue in this paper, addressing all the challenges faced in teaching software engineering, or informatics in general, requires a holistic approach. I see projects as the vehicle to bring all the disparate issues together and trade them off against one another. Of course, they should motivate the students more than the traditional classroom teaching. Of course, they should enable the students to learn by doing. But more than these reasons, a project-based approach should enable the students to apply system-level thinking, see technologies in use, and appreciate the difficulties and benefits of working with others in a team.

Course structure.

Tables 1-3 list the courses as envisioned for the three years of the bachelor curriculum. This is a draft proposal and it is likely that the exact set of courses will change somewhat as we gain experience with the curriculum. Also, I have not shown credit hours associated with each course as this will also change. The list, however, does give a flavor for the diversity of courses and the coverage of other disciplines.

Programming fundamentals
Computer architecture
Discrete structures I
Computer network architecture
Mathematics
Technology lab
Semester projects

Table 1. Courses in the first year

In the first year (Table 1), and every other year, the hours are divided roughly evenly between courses and the semester projects. Each semester has one semester-long project. The discrete structures course (continued in the second year) covers the theoretical background necessary for reasoning about discrete systems. The technology lab allows the student to learn about a particular technology. For example, if the student wants to learn about Linux or a programming language that is not normally covered in the curriculum, the technology lab provides a way to do that. Often, such courses can be taken on-line and the university does not need to provide customized courses for every available technology. This is one way that traditional universities can adopt the use of distance education to enhance traditional education.

Discrete structures II
Algorithms and data structures
Software design
Net-centric computing
Software development
Information and knowledge management I
Life sciences models
Technology lab
Semester projects

Table 2. Courses in the second year

Hardware and software co-design
The business of software
Information and knowledge management II
Technology lab
Economics and business models
Informatics elective
Semester projects
Final project

Table 3. Courses for the third year

In the second year (Table 2), there are two modules on software: one deals with software design and the other with software development. Traditional courses on software engineering in fact do not spend much time on software design because it is not an easy thing to teach. It is usually covered by discussing design notations. We have separated it into a distinct module to emphasize its importance. In fact, there are probably more general design (and architecture) issues than apply to just software. To be more specific, we have limited the module to software design. The intent is to examine examples of good designs and have lectures by good designers. The course on software development then covers the practical issues of software engineering, that is, tools and processes. Net-centric computing introduces a distributed systems view of computing. Traditional resource management issues of operating systems are covered in this course, along with communication and middleware. Information and knowledge management (continued in the third year) combines elements from programming languages, databases, artificial intelligence, and knowledge management. The purpose of combining them in a single course is to take an integrated view and see their common goals and models. Indeed, each of these subjects contains fundamental concepts and models that are important for every computer scientist. Unfortunately, it is not possible to require a complete course on each of these topics in a bachelor's degree program. By offering an integrated course, we intend to emphasize the different but related modeling and analysis approaches in these fields.

In the third year (Table 3), the course on the business of software deals with ethical and professional issues, as well as licensing models. It addresses both the role of software in business and the role of business in software. It is a way for the students to see a different big picture. The economics and business models course covers organizational theory and the role of information technology and systems in business. The informatics elective is a place-holder for different topics that are not covered in the curriculum. For examples, there are no required courses on compilers or graphics. This elective gives the student the opportunity to follow such a subject.

Master's degrees.

Clearly, three years are not enough to educate a complete computer scientist or software engineer. The Bologna convention calls for a two-year specialization master's program. The trend is for each university to offer several different specialized master's degrees, rather than a single master's degree in informatics. In Lugano, we have decided to continue our emphasis on interdisciplinary computer science in the master's program and offer master's degrees in "Informatics and …" This is, however, still under discussion and details are to be worked out. We have had many suggestions from people in different disciplines, both in industry and academia, that a combination of informatics and their discipline makes a lot of sense because informatics is such a central part of their business. For example, a master's degree in financial informatics could prepare a graduate who is conversant both in informatics and financial models and practices. The pervasiveness of computing and software has created a need for interdisciplinary informatics specialists. The master's degree is the right vehicle for providing this education.

7. Conclusions and open issues

In this paper, I have presented my view of the difficulties of educating a software engineer. Individual software engineers may use the problems I have mentioned as a guide or starting point for enhancing their background. But to institutionalize the educational experience, we need new curricula in universities. I have presented one such curriculum that is going to start in October 2004 at the University of Lugano. The curriculum is integrative, or holistic, interdisciplinary, and project-based. In some years, we will be able to report on the results of the curriculum and how it fares in comparison to more traditional curricula. In the meantime, there is room for other innovative curricula for software engineers because the problems of software engineering education are real, even if solutions are less clear.

For anyone interested in such issues, there are a few useful sources. Foremost is the curriculum recommendations proposed by ACM and IEEE. In particular, the two recommendations, one on computer science [3] in general and the other on software engineering [4], are full of ideas and arguments on how to teach individual courses as well as on choices of how to structure curricula depending on the size of the department, number of faculty members, orientation of the faculty, and so on. We have found the descriptions of the courses and the division into core areas and optional areas both inspiring and practically useful. The course contents take a modern view of computer science and clearly take into account recent developments in research and technology.

An interesting source of information is the website of the Career Space consortium (http://www.career-space.com). This consortium, consisting of many large multinational corporations involved in information technology, was sponsored by the European Commission to identify skill shortages, job profiles, and future needs of the European Union in information and communication technologies (ICT). This consortium also identifies behavioral (sometimes called "soft") skills as an important requirement for engineers and offers recommendations for structuring curricula with the use of a final capstone project. In the United States, the Software Engineering Institute (www.sei.cmu.edu), sponsored by the US Department of Defense, offers guidelines and courses on software engineering, primarily focused on practical matters.

Then there are of course textbooks on software engineering. A randomly picked example is [5]. One can find books with different emphases. Most try to be comprehensive and cover all the phases of the software process with some management thrown in. There are several factors that may be used to differentiate the textbooks. One is the view of object-orientation. Some assume object-orientation is the only way to do software engineering while others consider object-orientation as one of the approaches to software design. Another differentiating factor is depth versus breadth. Some try to cover all known techniques while others take a few techniques and cover them in depth. At one extreme, some textbooks consider only one specific approach, such as software engineering with UML or extreme software engineering.

Regardless of the abundance of textbooks and proposed curricula, one issue in software engineering education remains open and that is the question of how to teach software design. It would help the whole field of software engineering if we had a better idea of how to educate designers. This is of course not an easy matter since good designers exhibit abilities such as creativity, ingenuity, and good taste that are not necessarily teachable. On the other hand, we should be able to do better, perhaps by relying on other fields that have been teaching design for a longer time. We are planning to work with the Academy of Architecture in Mendrisio, Switzerland, to develop an interdisciplinary approach to teaching (software) design.

8. References

[1] Brooks, F. P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4 (April 1987): 10-19.

[2] Parnas, D. L., "The Professional Responsibilities of Software Engineers," IFIP Congress (2) 1994: 332-339.

[3] Computing curricula 2001, Computer Science, http://www.computer.org/education/cc2001/final/

[4] Computing curricula 2001, Software Engineering, http://sites.computer.org/ccse/

[5] Ghezzi, C., Jazayeri, M., and Mandrioli, D. Fundamentals of Software Engineering, Second Edition, Prentice-Hall, 2002.

[6] Parnas, D.L., "On A Buzzword: Hierarchical Structure," Proc. IFIP Congress 1974, Amsterdam, North Holland, 1974