The challenges of Software Engineering Education

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. da Vinci, 32, Milano, I-20133, Italy
carlo.ghezzi@polimi.it

Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. da Vinci, 32, Milano, I-20133, Italy
dino.mandrioli@polimi.it

ABSTRACT
We discuss the technical skills that a software engineer should possess. We take the viewpoint of a school of engineering and put the software engineer's education in the wider context of engineering education. We stress both the common aspects that crosscut all engineering fields and the specific issues that pertain to software engineering. We believe that even in a continuously evolving field like software, education should provide strong and stable foundations based on mathematics and science, emphasize the engineering principles, and recognize the stable and long-lasting design concepts. Even though the more mundane technological solutions cannot be ignored, the students should be equipped with skills that allow them to understand and dominate the evolution of technology.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]:

Keywords
Engineering, software engineering, education, models.

1. INTRODUCTION
We discuss the technical skills that a software engineer should possess. Rather than focusing on the organization of a course (or set of courses) titled "software engineering", we take the global viewpoint of a school of engineering and put the software engineer's education in the wider context of engineering education. We believe that software engineering (SE) should be viewed as an *engineering discipline*, and that a software engineer should be viewed as primarily an *engineer*. On these grounds, we analyze the principles and practices that pertain to the engineering culture and also what is special in the case of software engineering.

We emphasize that in general no engineering artifact involves the knowledge of a single discipline; thus, engineering disciplines cannot be taught in isolation but their essentials must be mastered by any engineer, independently of his/her particular field.

Another main issue that we face in engineering is learning by studying (at school) vs. learning by doing (at work) [1]. Our strong belief is that the two ways of learning are necessarily different and complementary. It would be a mistake to try to fake learning by doing at school, and it is almost impossible to resume the typical "learning in class and through textbooks" during everyday's work. However, it would be a fatal mistake to adopt the idea that the two worlds should ignore—or even disparage—each other. Learning at school should equip the students with all the fundamental skills that will enable them to proceed to lifelong learning at work.

In a continuously and rapidly evolving field like software, education should emphasize principles and recognize what are the stable and long-lasting concepts of the discipline. Even though the more mundane aspects of the state-of-practice (technology, tools, methods) cannot be ignored, the students should be equipped with skills that allow them to understand and dominate the various technological waves without being overwhelmed by them and thus avoiding the danger of rapid obsolescence.

In the following sections of this paper we explain in some detail why we believe in this approach and how it should be exploited in the organization of an engineering curriculum. The focus is obviously on teaching software engineering, but always keeping in mind that the software engineer's education is part of a wider process aimed at "forging an engineer".

Section 2 examines the fundamental skills of a software engineer, showing that they are shared by any engineer, but also pointing out important differences, which might generate challenges peculiar to the teaching of software engineering. Section 3 discusses how software engineering should be taught within university curricula. Section 4 focuses on "putting software engineering in context". Finally Section 5 draws some conclusions.

2. KNOWLEDGE AND SKILLS OF A (SOFTWARE) ENGINEER
Like any other engineer, the software engineer must master

- the theoretical foundations of the discipline;
- the design methods of the discipline;
- the technology and tools of the discipline.

In addition, he/she has to be able to

- keep his/her knowledge current with respect to the new approaches and technologies;
- interact with other people (often not from the same culture);
- understand, model, formalize, analyze a *new* problem;
- recognize a recurring problem, and reuse or adapt known solutions;
- manage a process and coordinate the work of different people;
- …

The above list, far from being exhaustive, is however sufficient to enlighten both the differences between learning by studying and learning by doing and the differences between learning these skills in software engineering and in other fields of engineering.

At the two extremes, certainly learning the theoretical foundations of a discipline is a typical school activity whereas much of managing a process and the psychology of interacting with people—whether one's manager, peer, employee, or customer—can be learned mostly through experience in the filed and exhibits

often sharply different requirements depending on the peculiar environment.

On the other hand, none of these skills—not even the above two extremes—should be learned exclusively in one way. Consider, for instance, the typical case of learning programming languages: some languages must be learnt in depth at school, but new ones will certainly appear and will have to be mastered at work. Therefore, teaching at school should focus on two related fundamental issues: how to apply good programming methods in any language, and how to understand the principles of programming languages. Such principles as program design via abstractions, modularization, decoupling, documentation apply to any programming language. Furthermore, a student who understands type systems, binding mechanisms, scope rules, runtime memory management, and other similar concepts, can more easily learn any new language and its specific subtle semantic features. It would be wrong to expose students to a plethora of different languages, "because they have to know all the possible tools they will encounter in practice". The other extreme, is exemplified by Dijkstra [2], who claims that initially students should learn only mathematical languages without running their programs on real computers. This, however, is equally counterproductive, and can mark a chasm between university and "the real world".

As another example, teamwork should be experimented in school project works. It is important that students, before they move to work, appreciate the need for scheduling their work with others in a team, negotiate requirements and specifications for the parts they are responsible of, etc.

In summary, learning at school should lay the foundations for all the skills we listed at the beginning of this section. The students should be mentored to learning most of what they will need to continue to learn after leave school, as part of their lifelong learning activity.

It is important to acknowledge that software engineering differs from the other more traditional fields of engineering in several specific points. These differences affect how is should be taught. Let us examine some typical cases.

- *Theoretical foundations are less mature, often "far" from being directly applicable, and hence less immediately useful.*
  This is at least a common feeling among practitioners, perhaps shared even by several academics. At a deeper insight, however, we could argue that, for instance, the difference between a Turing machine and a real computer is not greater than the difference between the notion of material particle (a body with a mass but no volume) and a real physical object such as a car, an aircraft, a fly, etc. or the difference between the notion of a "perfect gas" and the air of our atmosphere. The study of these abstract notions, however, is viewed in traditional engineering as foundational starting point that should be part of the engineer's education, whereas theoretical computer science is often neglected in the software engineer's education. We must acknowledge, however, that the models used in traditional engineering are more directly applicable, and hence more widely used by practitioners. As an example, consider how widely applicable Fourier transforms are in signal processing and in other fields of electronics.
- *Well-established models and notations are still lacking.*
  This point can be seen as the counterpart of the previous. Here too a striking example is provided by programming languages: it is somewhat astonishing that this field, which should be one of the most mature of computer science, still produces major novelties almost every year. It is even more surprising that these changes have an immediate impact on even lower-level (undergraduate or earlier) education. Conversely, introductory courses on physics still teach classical mechanics as it was taught decades ago. They would never try to incorporate new parts to teach quantum mechanics and relativity theory to freshmen.
  The situation is even more striking if we move to such aspects as requirements analysis, specification and architectural design, where the state of practice is still largely ad hoc, and methods, notations and tools are still largely ignored in the industrial world.
  The states of the art and practice, however, are evolving. The standardization and the increasingly wider adoption of UML have a positive impact on bridging several gaps. Not only UML has been a vehicle—certainly not perfect, but useful—to transfer some research results into the industrial world. But also it is an example of a notation originated in the field of computer science that is now recognized and adopted even in other fields of engineering. For example, it has been applied in logistics and in industrial manufacturing. This is not the only example of a technology that originated in our field and later was imported by other engineering fields. For example, control engineers now use automata-based discrete events models (e.g. Petri nets) along with their original continuous process models. We will come back to this issue in the following.
- *The distinction between the essential and the accidental complexity of software [3] is crucial.*
  This is the essence of engineering. When solving a problem, the engineer must neglect irrelevant details to focus on the essentials, otherwise he or she would be lost in an unmanageable complexity. The ability to separate concerns and focus on the relevant ones at each time is crucial for the engineer. This is quite different from the way mathematicians work. A mathematician works in a formalized world and mostly cares about precision of solutions. The hard question, of course, is what, how and when something can be considered to be a detail that can be neglected and what, how and when should one instead add more detail to achieve better modeling and analysis. The answer is difficult and context-dependent in most cases: one aspect could be irrelevant from some viewpoint and at some phase of the development, whereas it could be crucial in other phases.
  For example, in programming it is often useful to concentrate first on writing a correct program, ignoring such issues as performance and usability. These can be taken into account at a later point via correctness-preserving code transformations and by improving user interfaces.
  Other fields of engineering have some well-established guidelines to apply such difficult choices. In most cases they are rooted in the metrics of the adopted mathematical models: "neglect quantity X—say a current in an electric circuit—with respect to quantity Y if they differ of orders of magnitude". Metrics, and consequently any notion of continuity are lacking or highly controversial in most computer science models. Thus, the software engineer must often resort to more generic principles such as the separation of concerns to decide about what to focus on.

- *Mastering different approaches to manage project complexity.*
  The complexity of different software engineering projects, the diversity of application areas, and the lack of established common foundations makes it likely that the software engineer should master different methods and approaches and should be capable of choosing the best method and approach that fits the problem at hand. For example, the development process can follow a highly structured, top-down scheme (like in a waterfall lifecycle) or a flexible and iterative scheme (like in extreme programming). The process to choose depends on the size of the project, its criticality, the relationship with the customer, the stability of requirements, and so on. As another example, when a system is to be specified, different notations can be used according to the stakeholder's viewpoint. There is no universal process model and there is no universal notation. The engineer has to learn which to use, when, and why, depending on the problem at hand.
  Any engineer should own such a skill. However, the principles and, mainly, the techniques to confront with the various problems are much more established and less subject to fashion and "buzzwording" in traditional engineering than in software engineering. Examples can be easily found in the evolution from structured analysis/design, to object-orientation, to extreme and agile programming, etc. In parallel, and often in contrast, academia has been advertising for a long time formal methods, but those too have been subject to a rather uncontrolled and unpredictable evolution (thousands of abstract machines; logic, algebraic, and attempts to apply category-theory based approaches)[1]. It is certainly hard not to get lost in such a mess of more or less "revolutionary solutions". And it is hard to discern substance from pure cosmetics.
- *More emphasis is needed on interdisciplinary culture and communication skills.*
  This is due to the very nature of software applications. In most of them the software is the "intelligent glue" that integrates a complex heterogeneous system: examples go from embedded systems to industrial automation systems, to office automation, to virtual reality, etc. The software engineer, therefore, even more than others, must be able to understand problems and models not coming from his or her field and to interact with their specialists (of course, he will not replace the application domain specialist, nor will she have to gain the same depth and breadth of knowledge in their field).
  The above remarks, therefore, can suggest that, and explain why, the gap between learning by studying vs. learning by doing is deeper in software engineering than in other cases. Perhaps a consequence of this situation is that textbooks and courseware that aim at teaching practice are often wordy, purely descriptive, and overly informal. At the same time, textbooks and courseware that teach mostly theory are often unrealistic, only deal with toy examples, are not well understood and not well accepted by students (and instructors).

In summary, the widely practiced attitude of comparing software engineering with more traditional fields is still valid and thought provoking. It still shows important differences besides common needs and approaches. It also shows that often the younger discipline should strive to get closer to the better-established ones, but sometimes even the converse can produce important progress. Fortunately, much has been achieved since the birth of software engineering. But there is still a long way to go.

We wish that some of the recommendations that will be the object of the next sections could and should be addressed not only in organizing the teaching of software engineering but could also be taken into consideration, more generally, in engineering education.

3. CONSEQUENCES ON (SOFTWARE) ENGINEERING EDUCATION

In this section we articulate some general guidelines, derived from the analysis of previous section, on teaching software engineering *as an engineering discipline* [5].

- *Focus on lasting principles rather than last-minute fashionable technologies and buzzwords*[6], [7].
  This can be harder for software engineering than for other engineering fields because, as we discussed in Section 2, principles and theory may not be directly applicable, i.e., they do not yield normative practical techniques. Yet students should be motivated to learning them because they shape their mentality and make their approach to solving practical problems more mature and systematic. Because the recognition of the value of principles may come only later, even after years of experience, some "faith and trust" is needed from the students[2]. Furthermore, technology is evolving faster and often it is hard to distinguish between hypes and real new good approaches.
- *Integrate class teaching with projects.*
  This is a very critical issue in software engineering education. On the one hand, just studying principles on textbooks and even doing clever and insightful exercises is not real "learning" without a practical experience. On the other hand, mimicking the complexity of real-life projects in an educational environment can be impossible. Thus we need to find innovative ways of integrating project work in curricula (see, for example, [1] and [4]). We argue that projects should be realistic, but students should be aware of the differences with the real life, in terms of team size, duration and man power needed to carry over the project, requirements for compatibility with legacy systems, unavailability of "real" stakeholders, etc. At the same time, project work should exploit the opportunities (research methods, prototyping, …) that often are unavailable in the industrial world. Students may turn out to be carriers of innovation when they enter the business world.
- *Try to make things easy and understandable.*
  Stated in this way, such a recommendation may even sound obvious. How to achieve this goal, however, is far from trivial and involves conflicts. Indeed, many details should be abstracted away to make problems manageable within the typical terms of a university course and to help focusing

[1] This is another example of the big gap between learning (and teaching) at school and learning (and teaching) at work.

[2] It is not uncommon experience for us teachers to receive "post factum" recognition from former students with sentences such as "When I attended your course I hated topic xx. Also, as soon as I have been hired in my first company I was shocked by doing things that seemingly had nothing to do with what I learned at school. But now, after several years of practical work I understand and appreciate the value and usefulness of that teaching."

attention on the core of the problem, possibly one issue at a time. This may lead to class examples and exercises deal with "toy problems", a term that is often used in a derogatory sense, to mean unrealistic and ultimately useless or even misleading. We claim instead that toy problems may be quite insightful *if well chosen*; the literature is now rich of successful examples often carved from real-life problems but "cleaned up" from irrelevant and distracting details.

As a side remark, one might wonder why, in general, toy problems adopted in other fields such as mechanics are better accepted by students than in software engineering. For instance, why an exercise that asks for computing the trajectory of a missile suggesting to consider the missile as a material particle is appreciated and well accepted, whereas the classical dining philosophers problem is often disparaged as a "toy problem"—not only by students? Perhaps the answer to such a question is more of a psychological than of a technical nature.

- *Teach how to select and evaluate different methods and approaches rather than follow them like recipes.*

  Often methods force a normative behavior, but their application must be preceded by a careful and scholarly analysis of competitive approaches, a rigorous evaluation of the trade-offs and costs and benefits. As opposed to more traditional fields of engineering, software engineering has only a limited availability of ready-made normative methods that are scientifically supported. The ability to perform cost/benefit analysis, however, is a general skill that is typical and common to all areas of engineering.

  Here again we see how learning by studying and learning by doing should complement each other, the latter building on the foundations laid by the former. The teacher and the student should insist on a critical and comparative attitude; the teacher should encourage the student to experiment, to question claims given for granted in the literature, in order to come to her own opinions and decisions. The junior software engineer probably will not have so much freedom in making choices and will have to accept and comply with company-wide decisions made by someone else, maybe even long time before he or she has been hired. Nevertheless the engineer will certainly better exploit the company's methods if previously trained to apply a critical—yet constructive—attitude. The senior software engineer or the manager, instead, will have to choose a project's or even the company's new standards, and in this case he or she will *have to* be critical, comparative, and open to evaluating novelties by distinguishing the real progress from rubbish.

4. SOFTWARE ENGINEERING IN CONTEXT: REQUIREMENTS ON CURRICULA

Engineering is seldom highly specialized and narrowly focused. In most cases it deals with *systems*, often involving heterogeneous and interdisciplinary aspects; this is even more true in the case of software engineering. The ultimate purpose of software, in fact, is to allow computer-based systems to interact with their external environment, to control it, automate functionality, or provide service. The external environment may be the physical world of a controlled chemical plant or an intensive-care unit in a hospital. Or it may be the organizational world of a business unit to be supported in their sale operations; or a set of players who are competing in some Internet-based game.

Although a software engineer cannot be an expert in every physical domain, he or she must be able to interact with experts from those domains. Thus, *software engineering cannot be taught in isolation. It must be put in context*—how broad is an open issue.

We can categorize the context of software engineering as follows:

Mathematical background

This is a controversial issue although there is a general consensus that mathematics provides the fundamental background for every engineer. It is also true, however, that the gap between mathematical foundations and engineering applications is wider and less understood in software engineering.

It is often claimed that traditional engineering needs continuous mathematics whereas software engineering needs discrete mathematics. We think that such a view is fairly narrow-focused and misses the real relationship between engineering and mathematics. We claim, on the contrary, that *any* engineer, not only a software engineer, must have a solid background in all fundamental areas of mathematics:

- traditional continuous mathematics (differential and integral analysis and calculus[3]);
- discrete mathematics (logic, combinatorics, algebra; more generally, "non-continuous mathematics");
- statistics and probability theory.

There are many strong reasons to support our view.

First of all mathematics fosters rigorous reasoning. Each branch of mathematics, however, exhibits some specific forms of "building its own truths" which, *all together*, provide a formidable background for solid reasoning. Continuous mathematics stresses the unavoidability of errors but provides the ways to make them small enough so that they become tolerable. Combinatorics teaches how to "count" the members of a class of various elements and how to "arrange them according to some rule". Mathematical logic teaches how to build new truths from elementary ones and how to check whether a claim is true or false. Algebra teaches how to abstract and generalize particular cases into wider categories; Statistics and probability theory teach—among other technical skills—how to build reliable information on the basis of uncertain or variable data. All these skills should be part of the foundational tools of any engineer!

We should also be careful and flexible when we judge the "usefulness" of mathematics in terms of direct practical application. It is certainly true that many software engineers will never have to solve a differential equation. However, teaching only the mathematics that can be directly applied *now*, would be the same mistake as—say—teaching only the last minute middleware technology instead of the lasting principles of software construction. We simply cannot foresee what will be directly applied of what we are learning now; on the contrary we should be ready to learn, master, and apply some new technical and mathematical machinery whenever needed. The need for introducing some metrics in a new "domain of knowledge" and, consequently, the need to "make a distance as small as possible",

[3] Notice that we used both terms *analysis and calculus*: the former emphasizes reasoning and understanding mathematical definitions and properties; the latter focuses on building algorithms to solve practical problems once they have been formalized through suitable equations and a conceptual solution has been proved by some theorem.

which leads to the abstraction of a "continuous domain" and the concept of limit, may arise for the software engineer in several, often unpredictable, circumstances. For instance recent techniques in the security domain (e.g., SPAM filtering, misuse and anomaly detection) are based on some metrics defined on the message flows and identifying some threshold to separate the "good ones from the bad ones". Applications of continuous mathematics can also be found in other areas, such as learning algorithms or performance modeling and analysis.

Examples that show the need for software engineers to have a solid background in statistics and probability theory include software reliability modeling [8] and experimental methods applied to evaluating software quality, testing, and other empirical approaches.

We do not claim that any engineer must have the same strong background in all types of mathematics—it would be enormous and unmanageable. The relative balance between the various branches of mathematics can vary among the various fields of engineering. Also, advanced studies in some fields of engineering can require going in depth into fairly sophisticated mathematical technicalities. However, the fundamental background should be based on a wide range of mathematical fields.

Last, but certainly not least, the already emphasized need for the software engineer to interact with people with a different cultural background requires the ability to "find a common language" with them. This certainly applies to the mathematical part of the language, although it has even more important implications that will be discussed later.

The core of computing science

This is certainly the least controversial part, at least in its essentials. There is no doubt that topics such as programming and programming languages, computer architecture, operating systems, databases, networking, computation and complexity theory, etc. are an essential part of the culture of any software engineer.

Rather, there could be some discussion on the borderline between what is considered to be the core of computing science and what instead more properly belongs to software engineering [9]. As an example, does object-oriented programming belong to the basics or is it part of software engineering? What about testing? Certainly, one should learn some basics of testing even in introductory courses, but most testing methodologies belong to the more complex world of software engineering. Software processes and their management also share many organizational aspects with other disciplines: this is in fact another example of strong intersections, but also of striking differences, between software and other engineering disciplines. On the one hand, it is clear that in both cases we deal with the problem of organizing, managing, coordinating, and monitoring the work of several people; on the other side the human intensive distinguishing features of software production makes it much less predictable than, say, organizing the construction of a dam.

However, stating a precise borderline between the core of software engineering and its computing and organizational context is not such a big issue, once the important topics are well taught in a well planned and well coordinated curriculum.

The essentials of the "physical world"

Software intensive systems do not exist in isolation. They are built to be part of a wider and more general environment. Any (software) engineer should possess the necessary background to understand such a global environment. This includes:

- Mechanics, thermodynamics, electricity, etc.
- The technology on which computing science and engineering is built: electronics, telecommunications, …
- The "social physical world": economics, business organizations, communication, management, etc.

To some extent most of these disciplines are part of many computing curricula. However, their role, impact, and relative weight within curricula are highly controversial. Without going into a questionable quantitative evaluation, we claim that most often the teaching of the above topics is too superficial and descriptive. An even precise and accurate description of the physical phenomena is not enough to make their concepts and principles real *engineering tools*. Instead the typical engineering attitude that consists of

- observing a—not necessarily physical!—phenomenon,
- building a model of it,
- reasoning on it, both qualitatively and quantitatively, with the help of the model

should be emphasized and exemplified in some depth within the context of all these disciplines. We see several good reasons to do so, even sacrificing overspecialization in elective fields.

First, note that basic engineering principles—such as modularization, abstraction, modeling, verification, etc.—are quite general, and crosscut all engineering fields. They should be used systematically not only in teaching computing science topics, but also teaching "the essentials of the physical world". For example, one could stress the analogy between considering a component of an electric circuit as a *black-box*, of which only the external behavior is known, and the principle of information hiding, through a clear separation between module interface and its implementation.

Also, a broad culture helps reasoning by analogy and therefore fosters reuse. The ability of reusing any type of knowledge has been listed above as a major skill for the software engineer.

Once again, the need for the software engineer to interact with people with a different culture imposes the ability to understand and to master at a basic level their own models.

Let us also point out that here we are referring to the activity of formally reasoning on models of the real world. Such a real world is not exclusively the world of physical objects (plants, cars, aircrafts, etc.) but also the world of human organizations (banks, agencies, offices, hospitals, even courts of justice, etc.). These entities too have to be abstracted into a suitable formal model in order to be managed with the aid of software, which is formal by its very nature. This is not to say, however, that the activity of an engineer must be based exclusively on formal and mathematical reasoning on abstract models. Experience, common sense, intuition, … are fundamental tools of the engineer as well as of most other professionals. These skills, however, are acquired as a result of experience in the real world, rather than being learned by studying at school.

We would like to conclude this section by insisting that it is not only true that software engineering has to "learn" from traditional engineering, but also the converse is true. Software engineers cannot ignore continuous mathematics, but civil engineers, industrial engineers etc. cannot ignore discrete mathematics and

logic. The software engineer cannot be an expert in mechanics or thermodynamics, but rather should possess the basic knowledge that would allow him or her to interact with the specialists in these fields, should the need arise in acquiring requirements for a new application. Similarly, the mechanical engineer should be able to communicate with the software engineer who is designing the software embedded in the cruise control of a new vehicle, understanding the feasibility of certain real-time functionality. No engineer can ignore the basics of computer science, nor how software, which permeates practically all projects, is organized, documented and built: poor quality software can hamper a global system no matter how fine and sophisticated are other components. Notice that by "the basics of computer science" we do not mean how to use the Internet, productivity tools or, in general, computer tools, but the *principles* of computing science.

Cross-fertilization with other engineering fields is needed, in both directions; no matter how focused the curricula in the various fields are.

5. CONCLUSIONS

In this paper we presented our viewpoint on the challenges of teaching software engineering. We restricted our discussion to teaching within university education, ignoring the important areas of continuous education, special purpose intensive courses, and material mostly devoted to professionals. Rather than addressing *what* to teach in a software engineering class, we focused our attention on *how to organize* a curriculum for a software *engineer* and how to *put it in the proper context*.

We based our analysis on the unavoidable and complementary differences between learning by studying and learning by doing [1] and on the comparison between software engineering and other fields of engineering.

We concluded that a major challenge for the software engineering education comes from a deeper gap that exists between the two forms of learning than it happens in other, better established engineering fields. We analyzed the technical and "social" reasons of such a circumstance and suggested a few remedies (no "silver bullets", of course!). We also pointed out that the state of the affairs is evolving in a positive direction.

We strongly believe that, by its very nature, software engineering cannot be taught in isolation but must be put in the proper context. This is true for practically all branches of engineering, since most of the engineering tasks have to do with designing, building, and managing of *systems* that are quite heterogeneous in nature.

This remark broadens the scope of the challenge, because it leads to the question of what should be the foundations and the basic principles of engineering education, not just software engineering. At the beginning of engineering as a systematic discipline, it was fairly natural to define the core knowledge of any engineer. The continuous advances in science and technology inevitably led to a high level of specialization. For instance, presently our Technical University offers an order of 500 different courses for approximately 20 curricula that provide a degree in engineering. The mere proliferation of courses without caring about cohesion of the underlying foundations may lead to disasters.

We claim that the more the specialized culture and technology advance, the more education should strive to build a solid common mentality and culture that must serve as a foundation on top of which the special knowledge of a peculiar branch should be rooted and integrated with its context. The definition of such a common basis, however, should not be equated with keeping the same subjects that were taught for decades, before computing entered the engineering scene. Moreover, the common basis is not so much defined by identifying a common core set of disciplines, but rather a common approach that is based on building abstractions (models), reasoning about them and using the results to interpret the phenomena under study.

As we argued, the basics of continuous mathematics and classical physics should be still viewed as fundamentals in any engineering field. Moreover, since software permeates practically any engineering artifact and process, every engineering student should be exposed to the principles and the mathematical foundations of computing.

The challenges of education in a rapidly and continuously changing world should call universities, and in particular engineering schools, to rethinking their educational mission. Success (and even mere survival) in such a world requires identifying the long-lasting principles and strengthening the foundations. This is true for software engineering as for any other engineering fields.

6. REFERENCES

[1] Jazayeri, M., Education of a Software Engineer, Keynote presentation at Automated Software Engineering, Linz, Austria, 2004.

[2] Dijkstra, E.W., "On the Cruelty of Really Teaching Computer Science", *Communications of the ACM*, *32*, *12*, 1989, 1398-1404.

[3] Brooks, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, *20*, *4* (April 1987).

[4] Baker, A., Navarro, E.O., van der Hoek, A. "An Experimental Card Game for Teaching Software Engineering Processes", *Journal of Systems and Software*, to appear.

[5] Ghezzi C., Jazayeri M., Mandrioli D., *Fundamentals of Software Engineering, II edition,* Prentice-Hall, Englewood Cliffs, 2002.

[6] Abran, A., Seguin, N., Bourque, P. Dupuis, R., "The Search for Software Engineering Principles: An Overview of Results", Proceedings of the 1st Int.l Conference on the Principles of Software Engineering, Buenos Aires, Argentina, November 2004.

[7] Kramer, J. "Abstraction is Teachable?", Keynote at IEEE ACM SigSoft 16th International Conference on Software Engineering Education and Training; submitted for publication.

[8] Musa, J.D., Jannino, A., Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

[9] Parnas, D.L., "Software Engineering Programs Are Not Computer Science Programs", *IEEE Software*, *16*, *9*, 1999, pp. 19-30.