The Challenge of Teaching Foundational Ideas in a Modern Informatics Curriculum

Roland T. Mittermeir
Institut für Informatik-Systeme
Universität Klagenfurt
Austria

roland@isys.uni-klu.ac.at

Abstract: Teaching informatics at the university level implies teaching most recent advances in the field. Is there a space for including also concepts that constituted the foundations of recent methods and technology in a contemporary class? If so, why is it important to make students aware of dated but not out-dated ideas? And most of all, how can we motivate students to appreciate the currency of these ideas?

The paper addresses these issues from the vantage point of key concepts in software engineering.

1. Motivation

Teaching my software engineering class, I regularly sense a barrier of rejection in front of me when referring to the principle of information hiding [Parn 72] or even detail the concepts of structured analysis [SMC 74]. “Why should we remember this old stuff in the wealth of topics we have to master in this course anyway?” might be a rather benign question one can read from the student’s faces. “Has this old guy forgotten to update his slides? Today we write Java-Servlets and worry about web-programming!” might be in the minds of others.

The question is, how to deal with such reservations. First of all, we have to understand that they seem somehow justified. The students in front of me study informatics (computer science). Hence, they strive for education in a young and very vivid discipline. The laptops they have in front of them are of a recent generation. A five year old laptop would be considered already out-dated. They program in Java or in C#, modern object-oriented languages. They might not know exactly the date of conception of these languages. But it was not too long ago, that’s for sure. These modern languages are much younger than they are! They were born past 1980. Hence they might ask, what is the place for concepts conceived in the late or even early seventies? They are outdated! The storm of history and evolution has blown them away. We might accept that object-orientation has encompassed information hiding and made it to the standard of modern software development. Why worry about it? And how about SA/SD? That’s just historical reminiscence. Leave us alone with it![1]

Should we give in to such attitudes? Students are right to the extent that there are many valuable current concepts that find no room in the course. My in-class reputation would improve when focussing exclusively on them rather than on some of this old stuff. But there are reasons why I tend to be stubborn in my attempts making students aware of the roots of

[1]) The arguments mentioned are not particular to my students. As noted by Haigh, „computing students are disinclined to accept that study of particular examples now remote in time and space will teach them much of value“ [Haig 04].

some of the ideas that constitute the foundations of modern software technology. In the next section some of them are highlighted. If one considers the arguments raised there to be valid, the didactical challenge of how to motivate students who are younger than those concepts to appreciate their value has to be met. Some ideas towards this end are mentioned in section 3.

2. The role of “dated” foundations in current (SE) education

The motivation why to familiarize students not only with contemporary techniques but also with the roots of these techniques has many reasons. Without striving for comprehensiveness, a few of them are mentioned

- *General educational value:* From (future) academics we do not only expect that they are technically up-to-date. We also expect them to be “well educated”. Thus, it is important that they do not only know how to proceed in a certain situation but also why to proceed in the way they proceed. In German one refers to this as *Allgemeinbildung*. The root of this word, *Bildung,* is a holistic concept that can only be approximated by the English word “education”. A person with *Bildung*, i.e. *ein gebildeter Mensch,* can be considered as a human that incorporates vital elements of human culture. Culture, however, is a concept that rests on a long established evolutionary process. It rests on a combination of intellectual penetration and skilful expression of aspects perceived and (re)created according to the cultivated person’s perception.

  One might claim that in an engineering activity there is no room for such considerations. But this would be a far too short-sighted perspective on engineers and on engineering. An engineers work, hence also a software engineers work, has to fit into the context were the artefacts produced are to perform. To function properly, they have to interface well with the context into which they are embedded. For software this is a physical as well as a societal context. Societal context, however, is always also a cultural and, hence, a historical context. But wouldn’t it be strange to require understanding the historical context of the environment into which the artefacts one produces are to be integrated without knowing the history that brought our own discipline to its current level?

- *The historical trail as didactical avenue* was suggested recently by Böszormenyi [Bösz 05]. His claim is that teaching about the famous masters of our discipline would not only stimulate interest (teaching people about people) but that a didactical approach that follows the maturing of the ideas one wants finally arrive at will help comprehension. In this approach current techniques are not presented as still-image of the state-of-the-art but rather as the climax of a movie that presents the whole story how the state-of-the-art came about in a quick-motion.

  The arguments for this approach seem convincing. The problem is though, that it is quite time consuming. Hence, given the curricular constraints of engineering curricula one might not find the time and tranquillity to develop all key concepts in this manner. Nevertheless, the suggestion warrants further consideration.

- Becoming capable for dealing with *legacy software* is an argument that cannot be shelved away easily. While the arguments raised so far are founded on non-technical arguments, it is mandatory to empower students for maintaining and evolving legacy software. While COBOL will be hardly taught at university level CS-curricula, it is still a dominant language in legacy applications and hence in their maintenance. As there are very effective compilers, contemproary students are astonished about the efficiency of code generated by a good COBOL compiler [Kand 06].

But the issue is not that of a particular (dated) programming language. When need arises, graduates should be capable of learning a new programming language, even if this subjectively new language might look archaic from today's perspective. They will find books or other sources for learning the language. But the issue is to understand the legacy programmers reasoning. The driving ideas behind a design following SA/SD principles are quite different from the driving ideas behind a design following object-oriented principles. The programming language is strictly defined. Hence, learning on the job will lead to correct work. But given all the problems in maintaining legacy software, one should not expect that the young maintainer would also invest in learning the dated methodology. Hence, "training on the job" with high likelihood will result in a mixed-style solution. Thus, the system to be maintained by software engineers who do not understand the original principles of design will deteriorate in structure more than it would according to the general laws of software evolution [Lehm 80].

- Besides this, most concepts have also still *contemporary technical value*. E.g. the notion of coupling and cohesion introduced with structured design still holds when writing new software. It just needs to be interpreted in the proper way. Assuming object-oriented design, one has to decide on the proper structuring of state spaces. The argument that implementation objects stem from application objects identified during requirements analysis holds for the key objects of an application system. However, each system has further classes where the proper design of the state space rests with the designer. Too large a state space would lead to stamp-coupling, a notion discussed in [SMC 74] or at length at [YoCo 79]. Alternatively, one might reach a good design by reflecting on data base normalization theory [Codd 70, 72] as it was mapped to software engineering in [Mitt 91].

  Likewise, other principles of coupling and cohesion are still valid in class design, with the only side consideration that the state space managed by the various methods leads to a very special form of coupling.

- This shows, that the deep foundations provided by some now already historic seminal papers, sometimes need to be *transferred to the new problem spaces*. As example of such a transfer one might quote Dijkstra's "Go To considered harmful" [Dijk 68]. If students understand this concept not only verbatim but in principle, they should be capable of mapping it directly to data structures connected by pointers or even more general as a warning for producing entangled structures of any kind.

  But one might even stay closer to the original problem. Neat usage of exception handling constructs provided by modern languages will be easy to teach and to comprehend if one can refer to the goto-issue and the various kinds of problems caused by different versions of forward and backward jumps.

- Finally, *projection into the future* is an important issue. Educators should strive to familiarize students with durable concepts. In a discipline characterized by rapid technological change, this is a real challenge. Of course, even when considering that the professional life of current students will last into the next forty years, we cannot teach today those methodologies that will be current twenty or forty years from now. We have to stick to current knowledge. But we should empower students to master the shifts to be foreseen throughout the duration of their professional life. How can we do this better than by explaining shifts of paradigms caused by great ideas in the past? How can we do better than by showing them, that most of these ideas did not pop up out of the void but were rooted in the problems of their time and having them understand the technological milieu that helped these ideas to ripen?

With this requirement, which is based on the assumption that understanding our past will help us to properly shape the future, my arguments are somehow back at the initial argument. Appreciating the work that shaped our discipline has to be part of general education in order to get into the backpack of any computer scientist who wants to be considered well educated.

3. Attempts to overcome the motivational barrier

The arguments raised above, although only expressed in an exemplary manner, should suffice to strive for an approach that gives room for historic reflections even in a course focussing on most contemporary technology. But how to overcome the motivational barrier?

I consider three approaches that might help to achieve even the attention of those who expect from a modern informatics course to be taught just directly applicable technology.

a) *How did it start all about?* People interested in the mechanics of a device are usually interested in knowing how to get the thing started. Here, the issue is not starting up some device but the originating of a thinking process. Some investment is needed to have these students understand the transfer of ideas from starting an engine to conceiving an idea and have it mature. Nevertheless, this analogy can be built upon.

   The advantage of this approach is that it takes little time, since the transitory phase where the idea evolved can be traversed rather quickly. The disadvantage is that this is rather an ad-hoc approach. The individual concepts get rooted, but one might fear that the roots are not sufficiently deep for a given idea to network with other fundamental ideas that are somehow related to the concept just elaborated, e.g., because they matured in the same intellectual-technological biotope.

b) *Selected historic spots.* Alternatively, one might try to weave historic sprinkles into a highly up-to-date course. On the first glance, this approach appears to be even more ad-hoc than the approach to show the initiation of shifts of paradigms. It has, however, the advantage that less time is needed for the individual concept.

   If presented sporadically only, this form of reaching back into CS (or SE) history might go almost unnoticed by students. While this might be desirable for the educator's reputation as modern teacher, it is less desirable from the perspective of the overall educational aim. But if these historic sprinkles are systematically woven into the course, it offers the chance to set cross-references. This allows at least those who have fine sensory to realize that below the current wisdom of the discipline there is a network of fundamentals and a set of interacting trends. Hence, what is presented as individual colourful spot of historic evidence merges to a coherent concept. Thus, if persistently followed, this approach might well be superior over the approach showing for just a few concepts how they emerged.

   In comparison, one might classify the how-did-it-start approach as one striving for selective depth while a systematic historic-spot approach rather aims at covering the breadth of the course by some shallow historic underpinning.

c) *An explorative course.* With this approach, I'd finally acknowledge that it might be too hard for those who consciously lived already at the time some inventions crucial for the further development of informatics took place, to convincingly convey the merit of these former inventions to those who take such notions or even concepts built upon them for granted or even as already bypassed by the state-of-the-art. As contemporary witness, the "old" teacher might be ideal to tell an interesting story. But he or she is not sufficiently trustworthy in terms of the value one might currently attach to any deeper understanding of this idea. The still young student will use the state-of-the-art concept and considers this to be sufficient.

As the arguments raised above demonstrated that just using and just knowing state-of-the-art concepts and techniques is insufficient for many reasons, one has to strive for an alternative approach. To arrive there, I propose to separate the role of the contemporary witness from the advocate. Why not let the students themselves delve into the past and identify in a seminar-type course, where $concept_{hist}$ bears in $concept_{contemp}$. This course would require them to firstly, identify elements where the historical concept is directly applicable in a modern computing scenario. Secondly, the task might be to enlarge the scope and identify, where the historic concept still bears (and will persist to bear) outside of the concept that is generally considered as its direct ancestor.

The advantage of this proposal is apparently that it allows a leisurely approach. Consequently, it takes time and basically requires a distinct course. In special situations one might give related assignments in a conventional topics-based course. However, these assignments will require ample discussion and support from the side of the lecturer. Hence, I am doubtful whether anything that falls short of a distinct seminar on historic foundations of, in my case, software engineering (or any other well defined topic) will achieve the intentions defined in section 2. This however weakens this proposal to some extent as most places would relegate such a seminar to the pool of electives. Thus, only a minority of students will participate directly in this course. However, the material produced in such a course can be used in the actual topics course as supplementary reading material.

The three proposals mentioned can certainly be augmented by further suggestions[2]. They might point instructors to avenues how to achieve some historic grounding without giving in too much of their course-time and without loosing too much of the motivation and expectations students have with respect to a course intended to teach them contemporary skills. Which of the proposals is to be followed depends on the particular circumstances (and perceived mission) of the school and the curriculum where historic grounding is to be integrated.

4. Conclusion / Summary

Computer science topics in general and software engineering in particular are subjects that require teaching in a most timely manner. Nevertheless, there are convincing arguments that call for familiarizing students with the fundamentals of the ideas which constitute the contemporary discipline.

In spite of the arguments raised, there are substantial motivational barriers to be overcome before reaching the goal to ground modern methodology and concepts in their historic roots. The paper presents three approaches how these barriers might be overcome.

[2]) Lee is considering the issue from a broader, curricular perspective [Lee 04]. His proposals are distinct from the issues raised here in so far, as I am struggling with motivational issues of presenting roots of current notions in a general, topics oriented course, while the problems raised in this paper rather concern curricular constraints as well as problems of scheduling ("While you are away"-module) and staffing with qualified computer-science historians.

5. References

[Bösz 05] Böszörmenyi L.: *Teaching: People to People – About People: A Plea for the Historic and Human View;* in Mittermeir R.T.: From Computer Literacy to Informatics Fundamentals; Proc. ISSEP 2005, LNCS 3422, Springer Verlag, 2005, pp. 93 – 103.

[Codd 70] Codd F.E.: *A Relational Model for Large Shared Data Banks*; Comm of the acm, Vol 13(6), June 1970, pp. 337 – 387.

[Codd 72] Codd F.E.: *Further Normalization of the Data Base Relational Model*; in Rustin R. (ed.): *Data Base Systems*; Courant Computer Science Symposium, May 24-25, 1971, Vol. 6, Prentice Hall 1972, pp. 33 – 64.

[Dijk 68] Dijkstra E.W.: *Go To Statement Considered Harmful*; Comm. of the acm, Vol. 11 (3), March 1968, pp 147 – 148.

[Haig 04] Haigh T.: *The History of Computing: An Introduction for the Computer Scientist*; in: Akera A., Aspray W. (eds.): *Using History to Teach Computer Science and Related Disciplines,* Computing Research Association, 2004, pp. 5 – 26.

[Kand 06] Kandutsch H.A.: *Bubble Challenge: Über Legacy-Systeme, eine Gegenüberstellung neuer Technologien und ob Paradigmenwechsel in Softwaretechnologie zwingend Verbesserung bedeuten muss*; Seminar a. Software Engineering, Jan. 2006, Univ. Klagenfurt, 2006.

[Lee 04] Lee J.A.N: *History in the Computer Science Curriculum*; in: Akera A., Aspray W. (eds.): *Using History to Teach Computer Science and Related Disciplines*, Computing Research Association, 2004, pp. 33 – 38.

[Lehm 80] Lehman, M. M.: *Programs, life cycles and laws of software evolution*; Proceedings of the IEEE, 68(9):1060 – 1076, September 1980

[Mitt 91] Mittermeir R. T.: *Design-Aspects Supporting Software Reuse*; Chapter 11 in: Dusink E. M.; Hall P. A. V. (eds.): Software Reuse, Springer Verlag 1991, pp. 115 - 119.

[Parn 72] Parnas D.L.: *On the Criteria to be Used in Decomposing Systems into Modules*; Comm. of the acm, Vol. 15 (12), Dec 1972, pp. 1053 – 1058.

[SMC 74] Stevens W., Myers G., Constantine L.: *Structured Design*; IBM Systems Journal, Vol. 13, No. 2, May 1974, pp 115 - 139.

[YoCo 79] Yourdon E., Constantine L.L.: *Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design"*; Prentice Hall, 1979.