Discipline or punish: the cruelty of not teaching software engineering

Mark Priestley

3 October, 1994

I beg leave to throw out my thoughts, and express my feelings, just as they arise in my mind, with very little attention to formal method.

Edmund Burke, *Reflections on the Revolution in France*

When the term "software engineering" first came into wide use, it was not intended to name an academic discipline. The purpose of the first Conference on Software Engineering in 1968 [10] was to address the inadequacy of contemporary techniques of software development, the so-called "software crisis". It was recognised that many problems arose because the scale of the projects that were being attempted far outstripped the ability of people to manage and control such developments. Engineering was recognised to be a discipline that had successfully addressed the problems of constructing large artefacts, and the intention was to see how and to what extent the engineering approach could be adapted to help control the production of software. Since then, a large body of software engineering research and experience has grown up, and many universities now offer courses leading to qualifications specifically in software engineering, both at undergraduate and postgraduate level.

From the point of view of the software engineering industry, this should be encouraging. A characteristic of a mature engineering discipline is the existence of a consensus on the basic knowledge and skills of the discipline, or in other words, agreement about what ought to be taught to trainees. Typically, the education of an engineer comes in two phases: first, a period of academic study mastering the theoretical foundations of the subject, and usually culminating in the award of a degree or equivalent. This is followed by a period of apprenticeship when the application of the theory on industrial-scale projects is learnt. A relevant professional body will then usually certify the survivor of this process.

At the moment, the software engineering profession does not share these characteristics. In particular:

- There is no required, or even recommended, undergraduate qualification. Many computer science graduates get employment in software engineering, but on the other hand many companies are keen to recruit non-computer science graduates and to train them on the job.
- Software engineering curricula are very diverse, and there is wide disagreement about what they should include.

- The certification of software engineers is barely developed. The relevant professional bodies are weak, and certification is not widely recognised in industry as a necessary prerequisite for a career as a software engineer.

It might seem, therefore, that establishing software engineering programmes in universities would be unequivocally seen as a good thing, and as a valuable way of focusing the subject and speeding its evolution towards the status of a mature engineering discipline. A significant number of people involved in computer science and software engineering education do not share this view, however. There is a significant weight of opinion arguing that specialised software engineering courses should not exist, or at least not yet.

This chapter examines the reasons for this seemingly paradoxical stance, and argues that although there are genuine concerns underlying the arguments of the sceptics, it is very valuable to have specialised software engineering courses. Some proposals about the content of software engineering courses are then considered and the chapter finishes with an examination of the relationship between software engineering education and industry. Before this, however, it is worth looking at some of the arguments that have been put forward about the nature of software engineering and its relationship to other engineering disciplines.

1 Does software engineering exist?

As is perhaps inevitable for a young subject, software engineering regularly attracts misunderstanding and controversy. For example here is the eminent computer scientist Edsger Dijsktra on the subject, in a paper called "On the cruelty of really teaching computer science" [4]:

> ... software engineering should be known as "The Doomed Discipline": doomed because it cannot even approach its goal since its goal is self-contradictory. ... If you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has adopted as its charter, "How to program if you cannot."

Dijkstra's argument is based on a very personal view of what programming is, entailing among other things the belief that the task of writing computer programs is radically different from other practical and intellectual activities. His basic argument for this claim is the observation that the assumptions of continuity that we make about the behaviour of most materials and artefacts simply do not hold for software, which exhibits instead a radical discreteness. By "continuity" here is meant the property that small changes lead to small effects: for example, if I turn the volume control on my amplifier a small amount I can expect the volume of the output to increase by a correspondingly small amount. In general, however, software does not exhibit this property: very small changes to a program's code can have very large and drastic consequences. A classic example of this is NASA's first Venus space probe, which was reputedly lost because of the accidental omission of a single comma in the FORTRAN code written to control it. This story may be apocryphal, but it is not implausible.

Based on this property of the discreteness of software, Dijkstra argues that the mathematics appropriate to reasoning about software development is that based on logic and algebra, and not the classical calculus-based mathematics used by engineers in other disciplines. This is uncontroversial: there is very little evidence in the literature to suggest that anyone ever

thought that adopting the term "engineering" to describe software production entailed adopting all the detailed techniques and theories used by other engineers. (A prominent exception to this would appear to be David Parnas, initiator of some of the most promising ideas in the theory of software engineering. In a paper [12] whose eccentricity rivals Dijkstra's, he seems to be advocating a computing curriculum largely based on traditional engineering mathematics.)

More strongly, Dijkstra argues that programs are mathematical objects that can only be produced by means of a formal process resembling a particularly rigorous kind of mathematical proof. Revealingly perhaps, the process of teaching students to produce programs in this way is the "cruelty" referred to in the title of his paper. Further, Dijkstra clearly believes that such mathematical approaches are both necessary and sufficient to overcome the existing problems in large-scale software development. It is because software engineering attempts to address the problems of software development without adopting suitably mathematical techniques that he pronounces the discipline doomed.

The large-scale adoption of mathematical techniques in software development is often referred to as "formal methods". The proposal that this approach offers a foundation for software engineering is a radical one, and is examined in more detail later.

A more balanced view of software engineering is offered by Tony Hoare. In "Programming is an engineering profession" [7], Hoare contrasts "craftsmen" and "professional engineers" as producers of artefacts.

Craftsmen, who flourished in "earlier times and less advanced societies", possess unusual skills which have been learnt by imitation, practice, experience and trial and error in the course of a long apprenticeship to an acknowledged master of the craft. A craftsman

> knows nothing of the scientific basis of his techniques ... cannot explain how or why he does what he does ... and yet works effectively, by himself or in a small team, and can usually complete the tasks he undertakes in a predictable time scale and at a fixed cost, and with results that are predictably satisfactory to his clients. [7]

A professional engineer, on the other hand, not only serves the long apprenticeship of the craftsman, but precedes this by many years of formal study, covering the mathematical and scientific principles underlying his particular branch of engineering. As a result, he is equipped to give rational justifications of what he does, and is in command of techniques which allow very large projects involving many workers to be undertaken and successfully completed.

Hoare's purpose in drawing this comparison is to suggest that software development today has more in common with the craft model than the approach of the professional engineer.[1] By examining the transition from a craft-based industry to a mature engineering profession, however, we may begin to see ways in which we can hope to move towards a more satisfactory practice of software development.

2 Putting engineering into software development

In "Prospects for an Engineering Discipline of Software" [13], Mary Shaw considers in more detail what would be required of a mature software engineering discipline. She begins with a

[1] Given the references above to predictable time scales, fixed costs and satisfied clients in the craft model, even this may seem over-optimistic.

critical examination of the current state of software engineering practice, describing the term "software engineering" itself as a "statement of aspiration" rather than as a description of an existing discipline. She goes on:

> "Software engineering" is a label applied to a set of current practices for development. But using the word "engineering" to describe this activity takes considerable liberty with the common use of that term. The more customary usage refers to the disciplined application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance. [13]

The clear implication is that software development does not currently fit this description.

Shaw models the transition from a craft to a mature engineering profession as taking place in two major steps. First, demand for the product becomes sufficiently great to necessitate the introduction of large-scale industrial production rather than small, workshop-based enterprises. Establishing this larger scale of production involves codification of the techniques used, development of new techniques, and adoption of working practices applicable to the larger scale. The second step occurs when a science explaining the industrial practice emerges:

> The problems of current practice often stimulate the development of a corresponding science. There is frequently a strong, productive interaction between commercial practice and the emerging science. At some point, the science becomes sufficiently mature to be a significant contributor to the commercial practice. This marks the emergence of engineering practice in the sense that we know it today – sufficient scientific basis to enable a core of educated professionals so they can apply the theory to analysis of problems and synthesis of solutions. [13]

What stage has software engineering reached in this evolution? In Shaw's view, the introduction of software development methodologies in the 1980s marks the beginning of the transition from the craft stage to the commercial production of software. Colloquially, this is marked in the change of emphasis from "programming in the small" to "programming in the large". This transition has not universally taken place, however, and in many areas, and even in many commercial firms, craft models of production are still the norm. The science applicable to software development, on the other hand, barely exists, and only in very isolated cases are anything other than very informal techniques applied. Software engineering is therefore not yet a true engineering discipline, and at present is only emerging (slowly and painfully) from the craft stage. More optimistically, Shaw sees no reason in principle why a suitable scientific basis for software development should not in time emerge, and software engineering acquire the status of a mature engineering discipline.

3 Software engineering as a discipline

The relevance of the above discussion for education can be summarised in the following three points.

First, a characteristic of a "true engineering discipline" mentioned by both Hoare and Shaw is that a long period of formal education is a prerequisite for entering the profession. Assuming that software engineering aspires to be such a discipline, it is clear that university level courses in software engineering will be required to provide this formal education.

Second, in the case of the established engineering disciplines the purpose of this education is to equip neophytes with the scientific knowledge that underlies the field. The fact that this education is carried out in colleges and universities rather than in engineering firms is an indication of the maturity both of the field (education is more than simply apprenticeship) and of the science, which can be separated out from its applications and taught as a self-contained body of knowledge.

In addition to formal education, of course, a period of apprenticeship is still required before recognition as a qualified engineer is granted. The important thing about this, however, is that it takes place *after* the formal training: what is being learnt is the application of the principles of the discipline, not the complete subject.

Third, as Shaw says, the scientific basis of software engineering has not yet been developed. We can teach isolated bits of theory and a variety of useful techniques, but there is little sign yet of any unifying, underlying science that could be applied to the task of engineering software systems.

There is a clear incompatibility here between the desire of the software engineering profession to have an extended, formal period of education leading to professional accreditation, and the lack of scientific foundations for software engineering which might have been expected to form the major part of such an education. In its crudest terms, the dilemma for educationalists can be expressed as follow: if we have specialised software engineering courses, what are we to put in them?

In recent years, a vigorous debate has been taking place on this subject; a good source of material is the proceedings of the series of conferences on software engineering education run by the Software Engineering Institute of Carnegie Mellon University [5, 6, 2, 15, 14].

Participants in this debate tend to adopt one of the following positions:

1. Software engineering is too specialised a subject for undergraduate level education. Undergraduates should learn computer science, and become specialist software engineers only at Masters or PhD level.

2. In principle, undergraduate courses in software engineering are a fine thing, but at the moment the subject is much too immature to define and support such courses. In the meantime, computer science courses should be "seeded" with elements of software engineering.

3. Despite the immaturity of software engineering, it is still sufficiently distinct from computer science, and the needs of industry sufficiently pressing, to make it worthwhile offering undergraduate courses in the subject.

Implicit in these positions is grave uncertainty about the relationship between computer science and software engineering. The first position, for example, clearly assumes that software engineering is best thought of as a specialism within computer science; the second position does not go so far, but does assume that, pragmatically, software engineering can successfully be taught as such a specialism. Only with the third position is there acknowledgement of the possibility of software engineering being a separate subject from computer science.

The position argued for in this chapter is that there is a clear need to offer specialised courses in software engineering alongside traditional computer science courses. Although the two subjects share an interest in the same technology, and although there will be considerable overlap between software engineering and computer science courses, we believe that the needs

of the software industry and of future software engineers are best met by offering specialised courses in software engineering.

The first observation to be made in support of this claim is that it is not at all clear that a qualification in computer science is necessary to provide a solid foundation for software engineering. Software engineers clearly need a good appreciation of many aspects of computer science, but this can be gained in much less than the time available for a full honours degree, for example. Software engineering should therefore be viewed as a client discipline of computer science, not as a speciality within it. These claims are considered in more detail in the next section which examines the content of computer science courses in more detail.

Not only is computer science not a necessary foundation for software engineering, it is not sufficient either. As with any engineering subject, the ethos of software engineering is highly practical, and oriented towards the development of complex artefacts in response to real-world needs. This practical orientation is sufficiently fundamental to make it necessary to emphasise it throughout an entire university course: there is not time to acquire it otherwise.

A personal anecdote may make this point more vivid. Several years ago, I taught a final-year option course entitled "software engineering" to a group of students who up to then had followed a fairly traditional computer science course, similar to the one outlined in the following section. In the short time available, it was not possible to attempt any large-scale project with the students: the course instead attempted to convey the ideas of information hiding and modular design, and to outline some methodology of software construction. The results were predictably depressing: the students found the more theoretical ideas "abstract" and "irrelevant", and having only had experience of small, simple programs could appreciate neither the need for nor the benefits of a formalised software development methodology.

The most important conclusion that can be drawn from this experience is that the "software engineering" approach to software development can only be appreciated by students who have had prolonged exposure to software systems of a significant size, and who have experienced at first hand the difficulties of developing such systems in an unsystematic manner. It takes a long time, however, to give students this experience, and in particular it is not possible to convey both the problems and the solutions within the scope of a couple of specialised modules, as adherents of the second position above would advocate.

Finally, it should be recognised that there is a range of pragmatic and strategic considerations which have led to the institution of software engineering courses. For example, some institutions have felt obliged to offer courses titled "software engineering" in order to prevent the phrase being misused by non-computing departments. In a bid to appropriate the term, engineering departments are prone to interpret it as meaning "software *for* engineering". In this context, establishing a software engineering course can be seen as a clear statement of belief that there is a science of software development, and also as an attempt to provide a focus for the development of such a science.

Accepting the need for software engineering courses, however, makes the following question inescapable: while we are waiting for software engineering to attain maturity and identify its scientific basis, what material should (or shouldn't) be included in software engineering courses? The following sections chapter considers some answers to this question. Section 4 considers the relationship between software engineering and computer science, and the extent to which computer science is relevant to the software engineering curriculum. Section 5 evaluates the claim that formal methods form a suitable scientific basis for software engineering. Section 6 then examines a detailed curriculum proposal for undergraduate software engineering courses.

4 Computer science and software engineering

From the summary of the debate about software engineering education in the previous section, it is clear that there is a widespread belief that it is quite appropriate for software engineers to follow what is basically a course in computer science. This section looks in more detail at the computer science curriculum in order to evaluate this belief.

The name "computer science" is on the face of it rather curious: it implies the existence of a discipline centered around a piece of technology, rather than on any fundamental intellectual principles. Cynics might argue that equally compelling subjects could be invented, such as "refrigerator science"[2]: this would presumably cover all aspects of the construction and use of refrigerators, including electronics and cookery. Clearly, the coherence of computer science as a discipline has to be argued for.

A representative description of computer science may be taken from the report "Computing as a Discipline", published by the ACM Task Force on the Core of Computer Science [3]. This report splits the core of computer science into the following nine areas:

1. Algorithms and data structures.
2. Programming languages.
3. Architecture.
4. Numeric and symbolic computing.
5. Operating systems.
6. Software methodology and engineering.
7. Databases and information retrieval.
8. Artificial intelligence and robotics.
9. Human-computer interaction.

This list contains an impressive diversity of subjects, and taken at face value would seem to require considerable breadth of abilities and interest from the student. For example, a study of computer architecture (area 3) often requires at least an appreciation of digital electronics, whereas human-computer interaction (area 9) is based largely on practical applications of psychology.

Looked at more analytically, the list seems to conflate at least three subjects. Areas 1, 2 and 4 are concerned with the theory of computability and its applications, areas 3 and 5 with computer technology, and area 7 with information systems. The remaining areas are concerned with either the applications of computers, or aspects of the use of computer systems.

In this chapter, however, we are concerned not with the status of computer science, but with the role it should play in the education of software engineers. Any rigorous treatment of software engineering would be entirely contained in area 6, software methodology and engineering. Many of the other areas describe relevant knowledge: some areas, such as 1 and 2, are likely to be central in any software engineering course. Others, however, are

[2]This may not be a joke: given the way that higher education in the UK is developing, I would not put very much money on the non-existence of such a course.

either relevant only to particular application areas (such as area 8) or are areas where an appreciation of the constraints placed on the software engineer is all that is needed (area 3).

The conclusion of the last two sections, therefore, is that computer science as conventionally understood is neither a necessary nor a sufficient prerequisite for software engineering. There are significant areas of overlap between the two subjects, but it does not seem satisfactory to teach software engineering simply as an adjunct to a computer science course.

5 The formal methods research programmme

In recent years, the belief that certain areas of mathematics form the appropriate scientific basis for the study of software has received a lot of attention. This position is often referred to as a belief in "formal methods". Many things in computing have been described as formal, however: the characteristic of the position under examination here is the emphasis on treating software as a formal, mathematical object, amenable to mathematical techniques of manipulation, and with the possibility of achieving mathematical standards of rigour in software development. The applicable areas of mathematics in this context are believed to be logic and abstract algebra.

Such a view is in many ways highly appealing. It offers, for example, the prospect of being able to replace the messy, uncertain and above all boring business of testing by a process of proof, bringing with it greater certainty about the correctness of the software. The choice of underlying theory is also fairly compelling: it is plausible to treat programming languages as being formal languages in exactly the same way that logic is, and programs as being amenable to the same kind of manipulations as logical formulae.

There are signs, however, that the recent wave of enthusiasm for formal methods is ebbing slightly, and that formal methods have not lived up to the expectations of the enthusiasts. The title of a recent workshop presentation by Tony Hoare, "How did software get so reliable without formal methods?" [8], provides some circumstantial evidence for this claim. This title indicates an appreciation both of the limitations and marginality of current formal methods, and also of the achievements of current, informal, software engineering techniques.

There are many reasons for the failure of formal methods to live up to expectations, prominent among which are the three following:

1. The level of current languages. Perhaps the best known of products of the formal methods movement are formal specification languages such as Z and VDM-SL. These languages only achieve a very small amount of abstraction away from code. They abstract away algorithmic detail, but provide neither an effective way of discussing software structure nor a theory of software design. They are really at the same level as traditional third-generation programming languages, with the disadvantage of having well-understood and effective compiler technology replaced by the more difficult and ill-supported discipline of proof.

2. The inadequacy of current techniques. Many of the benefits of formal methods are held to flow from the mere act of specifying a system formally. Most authorities on the subject however at least pay lip-service to the importance of proof, or more generally, to the prospect of being able to formally derive a program from a specification and get a guarantee that it satisfies the specification. The complexity of the proofs required for

all but the most trivial problems is daunting, however, and current tools for supporting this activity are really only research prototypes.

3. Formal methods do not address what industry sees as the most significant problems in software development today. The typical formal methods development starts with a fixed specification of a self-contained system, and aims to prove the consistency of this system, and the correctness of a derived implementation. In much of the software engineering industry, however, this model of software is becoming obsolete: what is much more important is an emphasis on modularity, reusable components, distributed systems and interactive systems. Formal methods does not have a good record in these areas. For example, many attempts have been made to address the problem of retrieving software components from a library, based on a formal specification of their interface: none have reached any significant level of plausibility.

It is likely that at least some of these problems reflect the limits of current technology and the immaturity of the subject. Hopefully, further research will increase both the effectiveness and applicability of formal techniques, and they will come to play a more central rôle in software development. Some formal methods enthusiasts would like to claim more, however. The quotation from Dijkstra considered earlier indicated clearly his belief that formal methods can encompass the whole of software engineering. There are good reasons for doubting this, including the following.

1. Formal methods do not address the validation of specifications. A formal specification is a formalisation of a user's informal requirements, and it is hard to see how the process of checking that the specification meets the user's needs could be completely formalised, or that doing so would remove the phenomena of changing requirements, or of users changing their mind about their needs.

2. A significant proportion of software engineering is concerned with techniques for managing large-scale development, such as the study of design methodologies, project management issues, and quality assurance methods. Widespread adoption of formal methods would not do away with the need for these techniques: in an environment that developed software by means of proof rather than by coding, for example, there would still be a need for quality assurance procedures.

3. One response to the criticisms of formal methods listed above has been to emphasise the utility of formal methods in the restricted domain of safety-critical software, where there is a very heavy emphasis on reliability and correctness of code. Even in this area, however, theoretical objections have been raised. In [11] for example, it is argued that to obtain ultrareliable systems proof is not sufficient: it is also necessary to pay attention to the software's behaviour in its operational environment.

The topic of formal methods has been considered in detail because there has been an important and influential research programme in recent years attempting to establish formal methods as the underlying theory of software development. The discussion above can be summarised in the following three points.

1. Formal methods is an intriguing and promising area of research, and does capture some very important and characteristic aspects of software and programming. Further, there

is no other significant proposal at the moment which claims to provide a theory of the appropriate sort.

2. The current state of research in formal methods does not really allow us to make a judgement on the long-term prospects of the research programme. In particular, the lack of progress in mathematical modelling of large-scale program structure is disappointing: attempts to introduce modularity into Z and VDM for example have merely mimicked modular disciplines from programming languages, not explained them.

3. Even supposing the ultimate success of the research programme, it is extremely unlikely that software development will ever be reduced to the sort of formal exercise that Dijkstra seems to expect, and indeed long for.

Given the unresolved state of the debate, what rôle have formal methods to play in current software engineering curriculums? There is a strong tendency for software engineering courses at all levels now to include a mandatory element of formal methods. The problem is that the technology does not exist to make it possible to integrate them across the curriculum and so they are often perceived by students as being an interesting but ultimately irrelevant part of the course. The argument for inclusion is that even limited exposure to formal methods will raise the general level of awareness, and that as graduates with this exposure percolate through the industry the appreciation, and hopefully the use, of formal methods will increase.

6 A sample undergraduate curriculum

We will now turn from the rather abstract arguments given above and examine a concrete curriculum proposal for undergraduate courses in software engineering education. In 1989 an influential report was published by a joint working party of the British Computer Society and the Institute of Electrical Engineers [1], the two major professional bodies in the UK concerned with software engineering. The report drew the following main conclusions:

1. Specialist courses to educate Software Engineers require most of the time available for an undergraduate course.

2. Software Engineering requires a combination of engineering principles, design skills, good management practice, computer science and mathematical formalism.

3. The topics of *design* and *quality* should pervade the course, recurring in relation to many individual topics and creating an ethos in which students are enabled and encouraged to develop design skills, and apply engineering judgement in the selection and use of appropriate design tools and methods.

Reading the report, one gets the clear impression that the authors of the report see software engineering as having a much wider focus than computer science. This impression is borne out by the skills that the report specifies as being central to software engineering. There are seven of these "central skills", as follows:

1. system design, and the design of changes to systems;

2. requirements analysis, specification, design, construction, verification, testing, maintenance and modification of programs, program components and software systems;

3. algorithm design, complexity analysis, safety analysis and software verification;
4. database design, database administration and maintenance;
5. design and construction of human computer interaction;
6. management of projects which accomplish the above tasks, including estimating and controlling their cost and duration, organising teams and monitoring quality;
7. appreciation of commercial, financial, legal and ethical issues arising in Software Engineering projects.

These central skills are underpinned by seven "supporting skills": these are not specific to software engineering, but represent the fundamental skills and knowledge that software engineering students ought to possess. These skills represent necessary background knowledge: students should have a general appreciation of and competence in these areas, but it is not intended that students are expert in all these areas.

1. Information handling skills.
2. Mathematical skills, especially knowledge of discrete mathematics.
3. Knowledge of computer architecture and hardware.
4. Knowledge of digital communication systems.
5. Numerical methods.
6. Knowledge of some major existing components and systems.
7. Contextual awareness.

The curriculum is completed by a list of "advanced skills", which are intended to give the experience of depth. The are presented in the form of a list of topics: many of the topics would be equally at home on a computer science degree, and their specific relevance to software engineering and coherence with the rest of the curriculum is not made clear.

The first observation to be made about this proposal is that the list of "central skills" is similar to the list of "core areas" of computer science identified in the ACM report examined earlier. The major changes are an expansion of the "software methodology and engineering" area into the first two central skills, and the inclusion of project management and contextual issues into the BCS/IEEE proposal. Consequently, this curriculum also gives the impression of being rather a miscellany. This impression is particularly strong when the suggested list of advanced topics is examined. The intent of the authors here seems to be to pick topical and "state of the art" areas in computer science, rather than encouraging any depth of reflection into software engineering principles and practice

As argued above, in the absence of an underlying theory of software development this is to some extent to be expected. More positively, many of the suggestions made by this curriculum proposal are in broad agreement with the positions put forward earlier in this chapter. For example, many of the traditional topics of computer science are present at the level of "supporting skills", indicating that it is sufficient for a software engineer to have a

broad understanding and appreciation of the subject, without necessarily studying the topic in depth.

The pervading emphasis on design is perhaps the most positive aspect of the report. As the report says, design is perhaps *the* characteristic engineering activity, and it is precisely the design of large software systems that current computing theory has very little to say about. In the absence of an adequate theory of software design, however, the best that can be done is to offer examples of good practice, and plentiful case studies.

In summary, th BCS/IEEE report presents something intermediate between a traditional computer science curriculum, and a fully-fledged software engineering curriculum. Given the current state of software engineering theory, however, it is a very reasonable and helpful proposal.

7 Expectations of industry

The discussion so far has been largely from the academic point of view. There are good reasons for this: the status of software engineering as an academic discipline is still rather precarious, and the distinctive character and validity of the subject need to be argued for. However, the inchoate state of the subject also has profound implications for the relationship between software engineering educators and those in industry who will be employing the graduates of software engineering courses.

The majority of managers of software development projects have not had the benefit, or otherwise, of a software engineering education. This has two significant consequences. First, there is not a strong software engineering culture in industry today, nor any widespread adherence to what few principles of software engineering there are. Evidence for this is the overwhelming number of companies who have reached only the lowest level of the SEI software process maturity model [9]. Second, this means that industry has no clear idea of what can be expected of a graduate in software engineering, leading to many misunderstandings between academics and industrialists.

To caricature the situation slightly, industry tends to view the term "software engineer" as signifying little more than "up to date; programmmer proficient in the latest techniques and languages". Software engineering graduates should therefore be immediately productive in an industrial setting. Any disappointment in this scenario is seen as a failure of education.

Academics, on the other hand, are keen to stress fundamental principles, and the existence of "transferable skills". The fact that the graduate does not know language X is not a problem, but an opportunity: as a result of their education, graduates will have an understanding of the principles underlying the new language, and will therefore be able to learn and make effective use of it much quicker than would otherwise have been the case. Given the acknowledged rate of change in the computer industry, this ability should be seen as being much more valuable than mere expertise in a transient technology.

There is clearly room for a rapprochement between the two sides. Software engineering educators can play a twofold rôle in bringing this about. First, students need to be encouraged to keep the practical needs of industry in mind, and to be aware of the distinctive problems of real-life software development. As argued above, the ability to impart this awareness is one of the major advantages to be expected from specialised software engineering courses.

Second, there is scope for academics to make more use of the experiences of commercial software developers in software engineering teaching and research. There is a enormous

mismatch between the scale and ambition of the software being routinely produced by industry, and the extent to which such developments can be analysed or managed with the aid of software engineering theory. To a large extent, progress in software development techniques is currently being driven by industry, not academia. This leads to the ironic situation of academics fulminating against the inadequacies of industrial practice, while at the same time becoming increasingly reliant on the results of such practice. Word processors, the environment provided by windowing systems such as X windows or Microsoft Windows, and developments in networking such as the Internet form the background of every academic's life, yet none of these systems were produced with the aid of the kind of methodologies so beloved of software engineers: this is made clear in the description of Microsoft's working practices contained in [17], for example.

8 Conclusions

This chapter has argued that software engineering education is in a transitional state. There is wide, though not universal, agreement that software engineering courses as distinct from computer science courses are needed, and existing curriculum proposals still show very clear traces of their derivation from computer science.

It has been argued that the major influence hindering the further development of software engineering is the lack of an established scientific or mathematical theory of software which could serve as a foundation for emergence of a mature engineering discipline. This lack is reflected in the rather *ad hoc* way in which software development is still carried out.

An educational consequence of this lack of a scientific basis is that existing software engineering courses often give the impression of being merely anthologies of "best practice", and are sometimes hard to distinguish from computer science courses. Despite the undeveloped nature of the subject it is important that specialised software engineering courses are offered by universities, both to emphasise the difference of software engineering from traditional computer science, and also to act as foci for further research into the subject.

As the title of this chapter suggests, we are faced with a choice: to develop an adequate discipline of software engineering, encompassing both theory and practice, or to continue suffering the consequences of poorly understood and badly managed software projects. The establishment of distinct departments and courses of software engineering in universities can help this development, and hopefully have some effect on the industrial practice of software development, even in the absence of a developed theory of software.

About the author

Mark Priestley is head of the Software Engineering Division in the University of Westminster's School of Computer Science. His current research interests are in the areas of software design, formal methods and programming languages.

Farewell, Monsieur Traveller: look you lisp [16] and wear strange suits ...

William Shakespeare, *As You Like It*

References

[1] The British Computer Society and The Institution of Electrical Engineers. *A report on Undergraduate Curricula for Software Engineering*, June 1989.

[2] Lionel E. Demiel (Ed.) *Proceedings of the SEI conference on Software Engineering Education, 1990*. Springer-Verlag, LNCS 423.

[3] Peter J. Denning *et al.* "Computing as a Discipline". *Communications of the ACM* 32(1) 9 – 23 (January 1989).

[4] Edsger W. Dijkstra. "On the Cruelty of Really Teaching Computer Science". *Communications of the ACM* 32(12) 1398–1404 (December 1989).

[5] Gary A. Ford (Ed.) *Proceedings of the SEI conference on Software Engineering Education, 1988*. Springer-Verlag, LNCS 327.

[6] Norman E. Gibbs (Ed.) *Proceedings of the SEI conference on Software Engineering Education, 1989*. Springer-Verlag, LNCS 376.

[7] C.A.R. Hoare. "Programming is an engineering profession". In *Essays in Computing Science*, C.A.R. Hoare and C.B. Jones (editor), Prentice Hall (1989).

[8] C.A.R. Hoare. "How did software get so reliable without formal methods?" Presentation at Proof Club meeting, Edinburgh, March 21, 1994.

[9] Watts S. Humphrey. *Managing the Software Process*. Addison Wesley (1989).

[10] P. Naur (ed). *Proceedings of the First NATO conference on Software Engineering* (1968).

[11] Bev Littlewood, Lorenzo Stringini. "Validation of Ultrahigh Dependency for Software-Based Systems." *Communications of the ACM* 36(11) 69 – 80 (November 1993).

[12] David L. Parnas. Detailed reference to be provided.

[13] Mary Shaw. "Prospects for an Engineering Discipline of Software". IEEE Software, November 1990, 15–24.

[14] C. Sledge (Ed.) *Proceedings of the SEI conference on Software Engineering Education, 1992*. Springer-Verlag, LNCS 640.

[15] J.E. Tomayko (Ed.) *Proceedings of the SEI conference on Software Engineering Education, 1991*. Springer-Verlag, LNCS 536.

[16] Guy L. Steele. *The Common Lisp reference manual*. Digital Press (1984).

[17] James Wallace, Jim Erickson. *Hard Drive: Bill Gates and the Making of the Microsoft Empire*. James Wiley (1992).