

Modelling: A Neglected Feature in the Software Engineering Curriculum

A. J. Cowling

*Department of Computer Science,
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, United Kingdom
Email: A.Cowling @ dcs.shef.ac.uk*

Abstract

This paper argues that the concept of modelling, and particularly of software system structures, is not being given sufficient attention within current sources that describe aspects of the software engineering curriculum. The paper describes the scope of modelling as a general concept, and explains the role that the modelling of software system structures plays within it. It discusses the treatment of this role within the various sources, and compares this both with the experience of the role that such modelling plays in the undergraduate curriculum at Sheffield University, and with the practice in other branches of engineering. The idea is examined that modelling should be treated as a recurring concept within the curriculum, and it is shown that this gives rise to a matrix structure for the software engineering curriculum. The paper discusses how such a structure can be mapped into a conventional hierarchical curriculum model, and the relationships that need to be made explicit in doing so. It describes the practical implications of these results for the structures of degree programmes in software engineering.

Keywords

Software engineering education, software modelling, software engineering theory, software engineering practice, software processes, curriculum structure, degree programmes.

1. Introduction

As its title suggests, the motivation for this paper is a concern that the process of curriculum development for software engineering (SE from now on) is in danger of neglecting the topic of modelling, and particularly the modelling of the products that SE is concerned with developing. The purpose of the paper is therefore to examine the role that this topic should have, both to demonstrate that there is genuine cause for this concern, and to suggest a remedy for it.

The starting point for this is to define, in section 2, the scope and organisation of the topic of software modelling, and then in section 3 to examine the role that it currently has, as evidenced by various published sources, and in particular the Guidelines for SE Education [1], where it appears as a "recurring concept". Sections 4 and 5 then compare this with the role that it should have, based respectively on the experience gained from the undergraduate degree programme in SE in the department of Computer Science at Sheffield University, and general practice in other branches of engineering. To see how software product modelling can be given an more appropriate degree of emphasis, section 6 reviews the significance of treating it as a recurring concept, while section 7 shows how the conclusions of this might be accommodated within the

models that are being developed for the SE curriculum. To bring out the practical relevance of what is otherwise a fairly theoretical discussion, section 8 examines some of the implications of this for SE degree programmes, and section 9 summarises the conclusions of the paper.

2. The Scope of Modelling in SE

As noted in the introduction, the SE Guidelines provide an obvious starting point for defining the scope of the topic of software modelling, in that they identify a set of seven topics that are called "recurring concepts", a term that is borrowed from Computing Curricula 91 [2] (CC91 from now on). One of these seven recurring concepts is entitled "Software Modeling", and the definition for it that is given within the SE Guidelines is as follows. "The Software Modeling component covers principles and methods for modeling software architectures and software development entities. This includes techniques for using abstraction, modularity and hierarchy to model software functionality, data object relationships, behavior models, and formal methods." In practice, this set of topics can be broken down into more detail in a hierarchical fashion, as illustrated in figure 1, where there are two important features to note.

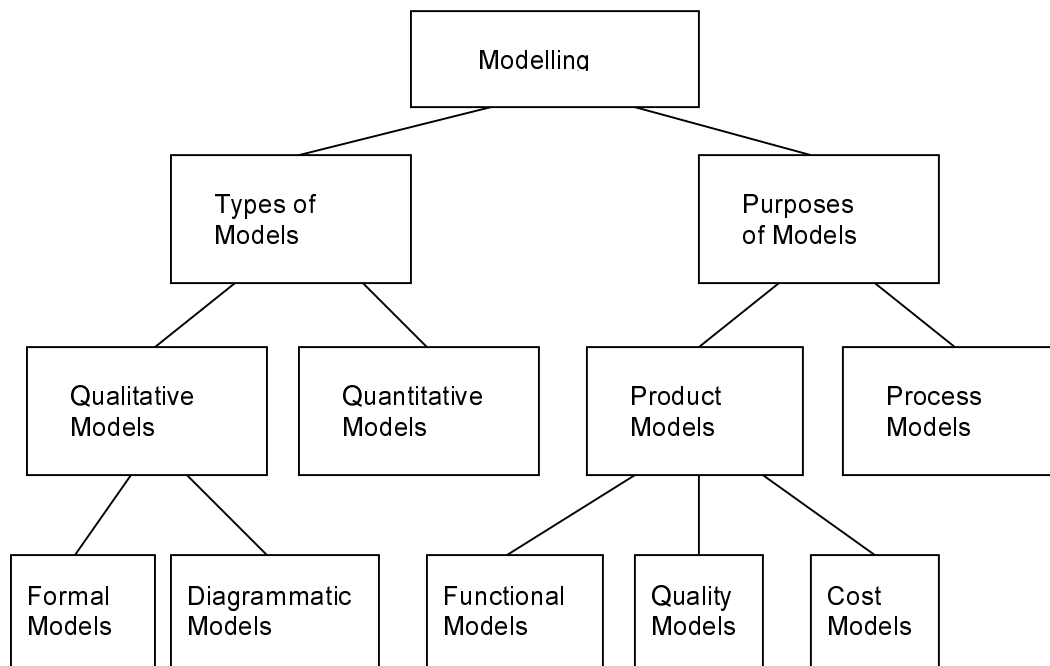


Figure 1: A Hierarchical Structure for Modelling in SE

One feature is that the types and purposes of models are orthogonal, so that both product and process models will each have qualitative and quantitative forms. Of these, the qualitative or structural models are concerned with describing the different kinds of concepts (eg modules, functions or "data objects") that occur in the structure being modeled, and the various relationships between them (such as "hierarchy") that go to make up this structure. Such models can then be expressed either diagrammatically using notations like UML [3] or its predecessors, or formally using notations such as Z [4], VDM [5] or B [6]. By contrast, quantitative models are expressed in terms of equations relating different properties of the objects being modeled,

although usually the forms of these equations will then be derived in part from the underlying structures that are described in a qualitative model.

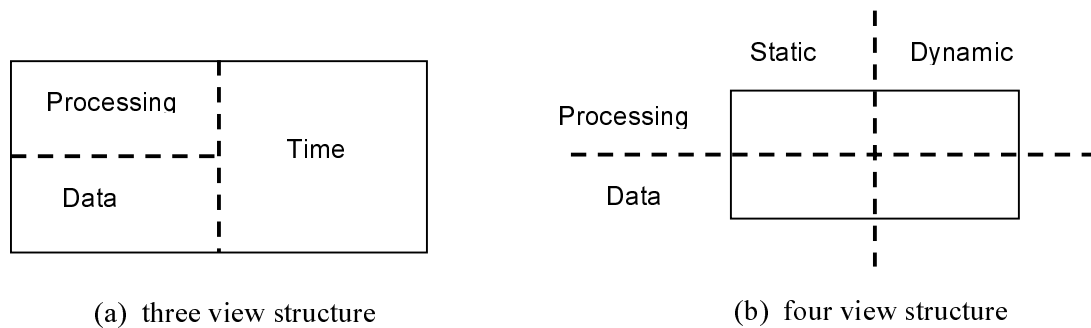


Figure 2: The three-view and four-view structures for models of system functionality

The other feature is that the product models divide into three main groups, covering the aspects of functionality (where the term is used here to also include the underlying "data object relationships"), quality (which partly relates to the aspect of "behavior"), and the cost of construction. Of these, the qualitative functional models form the basis for the other two groups, and conventionally they are divided either into either three or four groups, as illustrated in figure 2, although an important insight from formal methods is that these are actually not different models, but simply different views of the parts of a single integrated model (represented by the outer rectangles in the figure). The older structured analysis and design methodologies identified just three of these views, as shown in figure 2(a), but they referred to them as separate models, namely data models, processing models and time (or behavioural) models. More recently, many of the object oriented methodologies have recognised that the last of these three effectively splits into two parts, one concerned with the behaviour over time of the data for a system, and the other concerned with the behaviour over time of the processing that it performs. This, therefore, gives a division into four views, in which each of the data and processing aspects are divided into a static part and a dynamic (behavioural) part, as shown in figure 2(b).

3. The Current Role of Modelling in SE

To see the extent to which software product modelling is being neglected within the SE curriculum, one starting point is the set of common introductory textbooks on SE. This is where the project to construct the SE Body of Knowledge [7] (SWEBOK) started, and a check has been made on the index and contents pages of a large part of their sample, plus a few others, to determine what references they make to this topic. With a couple of exceptions (both published in 2001), if there is any mention of modelling in them it is typically referring either to process models, or to modelling non-functional aspects of systems such as cost or reliability. In general there is certainly no suggestion that either modelling itself as a concept, or the application of it to the structures of the products being created, should be treated as a coherent topic within SE. Even in the two newest texts, which do introduce modelling, the coverage focuses primarily on its role within requirements analysis, rather than as a concept that underpins the whole of SE.

To be fair to the authors of the older texts, this could be partly a problem of terminology, since related terms such as abstraction, formal methods or different types of diagrams all appear, and point to material that is essentially about aspects of modelling. Also, it could be argued that at least some of these textbooks were written originally for readers who already had a significant knowledge of basic computer science (CS from now on), and so who might be expected to be

familiar already with at least some of these concepts of modelling. Nevertheless, the effect is that the coherent treatment of the concept itself is still missing, and particularly insofar as it relates to activities such as specification, design, construction, validation or verification.

This effect has then carried over into some of the documents that define models for aspects of the SE curriculum. Since the SWEBOK project started with a similar sort of analysis of textbooks, and also of the ISO/IEC standard 12207 [8], it is not surprising that the Strawman version of its guide made little reference to modelling, and this is still the case with the current version [9]. Indeed, the term does not appear anywhere in the chapter headings or sub-headings of the guide, and one has to go down to the overviews of each chapter in the introduction to the guide to find any references to it, and these are only in the context of conceptual modelling within requirements analysis (in the overview of chapter 2), models for software metrics (in the overview of chapter 8) and process models (in the overview of chapter 9).

Similarly, the structure of the SE knowledge area for the CS volume of Computing Curricula 2001 [10] (CC2001 from now on) follows much the same pattern, since it is based on the same set of sources, together with the Guidelines document. This structure did, however, have to reduce the material heavily to fit into a single knowledge area, so that many topics had to be condensed or even omitted. Consequently, modelling does not appear in the titles of any of the twelve knowledge units that make up this knowledge area, although it does appear in some of the topics that, but not with as much importance as activities such as requirements analysis or design, or others of the recurring concepts from the Guidelines document.

Most recently, the first stage in developing the SE volume of CC2001 has involved producing a definition of the model known as SEEK (SE Education Knowledge), and the first public draft of this [11] had just been released at the time that this paper was being written. This has a wider scope than the SWEBOK guide, since it also needs to describe foundational material that is required to support SE, as well as the core SE material. For the latter, though, it follows a very similar structure to the SWEBOK, which it explicitly acknowledges as a key input. The result is that its coverage of modelling is similarly weak, although there are more references to it. Thus, "Modelling" is one of four units in the "Fundamentals" area, "Requirements modelling and analysis" is one of six units in the "Requirements" area, and several other units have aspects of modelling as one of their topics, such as "Metaphors and conceptual models" as one of the topics in the "User interface design" unit, and "Modelling and specification of software processes" as a topic in the "Process concepts" unit. While this is an improvement, it still does not bring modelling up to the level of importance attached to topics such as software processes or software quality, which both form complete knowledge areas.

4. Modelling and the Sheffield SE Programme

This limited coverage of modelling in these sources contrasts with the experience of the author and colleagues in developing and delivering an undergraduate degree programme in SE. This started in 1988, and its early development is described in [12], which emphasises the role of regular reviews of its goals and curriculum. As highlighted in section E.1 of the SE Guidelines, and as discussed in a separate paper [13], these reviews have increasingly focused on the skills that the students should develop within the programme, and particularly the overall skill of being able to develop software systems. The best way of achieving this is through projects involving the development of actual software systems, particularly if these can be real systems for real clients [14]. The most recent major revision of the Sheffield curriculum (its fifth) has therefore been based on the principle that its "spine" should be formed by a sequence of major projects, one in each academic year, with the rest of the curriculum being driven largely by the need to ensure that the students have acquired the knowledge needed for undertaking these projects.

Since this programme follows the standard UK model that full time bachelor's degrees take three years, it is the first year that gives students their basic introduction to SE (rather than the second year, as it might be in the North American model), and hence lays the foundation for their ability to develop software systems. This first year is structured round a project that is carried out in teams, and is based on fairly simple scenarios – typically requiring three to five business classes, and a similar number of relationships – with individual lecturers playing the roles of clients in order to give a variety of approaches to each scenario. The process for these projects is a standard "waterfall" progression through a sequence of five stages: requirements analysis, formal specification, design, implementation and testing. Each stage involves the production of an appropriate deliverable, and a key feature of the projects is that, after each stage, each student team passes its set of deliverables on to another team who have not previously seen that scenario, so that the new team will work on it for the next stage.

From experience of running these projects, it has become apparent that at each stage the primary focus of the teaching has to be on equipping the students to produce the technical content of these deliverables. This technical content largely consists of the models of the systems under development that need to be produced at each stage, which are respectively: a conceptual model (mainly UML use cases and a class diagram), a formal model (in Z), a design model (mainly UML sequence diagrams), the code itself (in Java) and a functional testing model (a set of test cases). Thus, at least in the first year of this programme, the bulk of the curriculum content for core SE consists of various aspects of modelling, and the other topics that are fitted round this are mainly ones which appear in the "Fundamentals" area of the SEEK.

It has also been our experience that progressively less formal teaching time has needed to be spent in this first year on aspects of software development processes, as once the basics have been introduced, the students essentially learn the rest by doing it in the project. Much the same is true of software quality and software tools, and in particular our experience of trying to utilise software tools for these projects has been very mixed. It has given the students valuable insights into the contributions that tools can make, but the difficulties involved in using them have often consumed a disproportionate amount of time compared to the benefits gained from them.

In the subsequent years, though, the emphasis does have to switch more towards processes, since the projects that are carried out in the second year use external clients, and as a result are much more varied, so that the issue of how to adapt the process to suit the constraints of different projects becomes much more important. This has been particularly true in the last year or two, where these projects have also been the subject of pilot studies in the newly-formed Sheffield SE Observatory. The aim of these studies has been to establish the methodology for experiments comparing agile processes (namely extreme programming) with more traditional ones, and so the curriculum has needed to cover a fair amount of material relating to these different approaches to SE processes, in order to set the scene for these experiments [15]. Even so, the more general concepts of product modelling still form a significant element in the second and third years of the curriculum, underpinning as they do the coverage within the relevant modules of topics as varied as design patterns, database structures and software cost estimation, all of which depend to some extent on models of software system structures.

5. Modelling in Other Branches of Engineering

Given this difference between our experience, that modelling of the artefacts to be developed is a key feature of the SE curriculum, and the coverage of this kind of modelling within formal SE curriculum structures, where it plays a much less significant role, it is appropriate to compare this situation with practice in other branches of engineering. In preparing this paper a simple piece of research was done, consisting of a series of keyword searches of the university library

catalogue, using the keywords “Introduction”, “Engineering”, and then the names of different branches of engineering (civil, electrical, mechanical and chemical). From the results of each search, a small set of textbooks were selected which appeared to be representative of that branch of engineering, trying in fairness to avoid ones whose titles suggested that they would in any case be strongly oriented towards particular kinds of models. The contents of these were then examined, looking primarily at chapter headings and the topics that they covered, to see how (if at all) they treated modelling.

The results of this were that ten out of the twelve texts all had the same characteristic, that their early chapters are almost entirely concerned with describing and analysing the fundamental models used in that branch of engineering for the materials, components and systems with which they are concerned. The two exceptions were a text which (despite the title “Introduction to civil engineering construction”) actually turned out to be not for engineering students at all, and another (with the title “Civil engineering practice: an introduction”) that was essentially about professional issues, which means that neither of them could be regarded as representative.

These conclusions for the importance of modelling were then validated by a series of discussions with colleagues from other engineering departments in the University of Sheffield, who all confirmed that they see the introduction to their branches of engineering as having to centre on the combination of theory (as represented by these models) and practice. Also, they particularly confirmed a feature that had been noted in examining the textbooks, namely how often the phrase “theory and practice” occurs in their chapter headings. This is hardly surprising, since arguably it is the combination of these two aspects that is the most fundamental characteristic of engineering. Despite this, though, one will search almost in vain for any occurrence of the word “theory” in the contents pages of the standard textbooks on SE, or in their indexes either, with the exception of a few references to Boehm’s “Theory W” [16], which has only very limited relevance to the issues being discussed here.

6. Modelling as a Recurring Concept

While this clearly confirms that the concept of modelling ought to play a more important role in the SE curriculum than it currently does, it also creates a problem for those trying to develop models of the curriculum, namely that of how this concept should be fitted in to these models. As has already been indicated, the SE Guidelines tried to solve this by adopting from CC91 the idea of recurring concepts, and describing modelling as one of these. Unfortunately, though, this idea of recurring concepts does not seem to have been generally accepted, in that it has not been used in the structure of the SWEBOK, while the CC2001 project seems to have abandoned it altogether, for there is no mention of it as a structural feature in the CS volume, and it has not so far been utilised in the SE volume either.

This is regrettable, as it has been shown elsewhere [17] that the idea of recurring concepts can be valuable in those aspects of a curriculum that are dependent on material from related disciplines. On the other hand, the idea of recurring concepts is not an easy one to work with, because concepts can recur for a variety of reasons, and the fact that these have not previously been analysed may have contributed to the idea falling out of favour. To try to remedy this, the way in which the idea was used in the CC91 model has been reviewed, to analyse how the various recurring concepts can be classified. The definition that it used of a recurring concept was that it should have three characteristics: it should occur throughout the discipline, it should have a variety of instantiations, and it should have a high degree of technological independence. This gave a very mixed set of twelve concepts, and the analysis suggests that these can be classified into four categories, depending on why they possessed these characteristics and hence their impact on the structure of the curriculum. These four categories are defined in table 1.

Table 1. Proposed categories of recurring concepts.

Category and Examples	Structuring Role
Emergent CC91: consistency and completeness, ordering in space, ordering in time.	The concept appears in different forms in different topics, and only serves to integrate the delivery of these topics.
Applicable CC91: complexity of large problems, security, reuse. SE Guidelines: ethics and professionalism, tools and environments, documentation.	The concept forms an identifiable topic, and other topics then apply it.
Structural CC91: binding, levels of abstraction, tradeoffs and consequences. SE Guidelines: software quality, software metrics.	The concept forms an identifiable topic, which shapes the treatment of other topics that apply it.
Foundational CC91: conceptual and formal models. SE Guidelines: software modelling, software processes.	The concept underpins the basics of many other concepts.

The significance of the foundational recurring concepts is that they play nearly as important a role in structuring a curriculum model as do the basic knowledge areas. Indeed, in the case of the "software processes" concept, this role in the SWEBOK, CC2001 CS and SEEK models is that its structure as a set of activities does not merely recur across their sets of knowledge areas, but provides the very basis from which many of them have been identified. By contrast, the foundational role of "software modelling" has not had the same impact on structure, but it is shown by the fact that each of these activities within the process can be described essentially in terms of constructing, transforming or validating product models, so that the structure of this concept (as described in section 2) does not merely recur across topics derived from the concept of software processes, but is orthogonal to them.

7. Modelling and Curriculum Structures

This property of orthogonality means that the underlying structure of the SE curriculum is really a two-dimensional matrix, with dimensions for software modelling and software processes respectively. Indeed, it has been argued elsewhere [18] that there are really three foundational concepts for SE – products, processes and people – which would give a three-dimensional structure. Fortunately, the role of people can be largely separated into two parts, as users of the products and as developers within the processes, so that by factoring out issues of ethics and professionalism the structure can be reduced to a two-dimensional one. This, though, still leaves the problem that the most convenient format in which to document a curriculum model for human readers is the one-dimensional one of a list or set of knowledge areas, each composed of knowledge units and topics. Hence, the challenge is to reduce this matrix structure, based on product models and processes, to the form of such a linear hierarchy, but without losing the essential relationships between these components that derive from this underlying matrix.

Solving the first part of this challenge is straightforward, since it essentially involves selecting one of the dimensions of the matrix and using its decomposition to identify what we might call a

primary set of knowledge areas, and then adding to this a secondary set to cover major topics that would otherwise be factored out by this process. This is exactly what all of the curriculum models for SE have done, using the process dimension as their basis for the primary set of knowledge areas. It might be an interesting academic exercise to try to develop an alternative model by starting instead from the product dimension, but such an exercise is outside the scope of this paper. Instead, it is more helpful to focus on the problem that has not been properly solved yet with the current models, which is the second part of this challenge, namely that of representing properly the relationships between their knowledge areas and the orthogonal topics that derive from the software product modelling dimension of the underlying matrix.

In considering how this second part of the challenge might be solved, there is a useful analogy that can be drawn with the two orthogonal topics of data structures and algorithms in CS. This is based on the observation that software product models effectively describe the structures of the data that is created and used in the activities that make up SE processes, while these different activities can then be seen as describing (in a more-or-less algorithmic fashion) how this data is processed. The way in which this orthogonal property of data structures and algorithms is handled in the CS volume of CC2001 is by defining the foundational concepts of each as separate knowledge areas in their own right, namely "Discrete Structures" and "Programming Fundamentals". Then, the material where they inter-relate is formed into a separate area, namely "Algorithms and Complexity".

Scaling this solution up to the orthogonal relationships between software product models and SE processes suggests strongly that the foundational concepts of each should form separate knowledge areas. Then, the material where they inter-relate is, of course, precisely what has been referred to above as the primary set of knowledge areas. Currently the existing models go part way towards adopting this solution, in that they all include a knowledge area for SE processes, reflecting the fact that it is a foundational recurring concept. As yet, however, none of them have included an equivalent knowledge area for software modelling, as the other foundational recurring concept, and the whole argument of this paper leads to the conclusion that such an area should be included.

A further step, which would help to make the relationship between products and processes clearer and more explicit, would be to make a separation between two types of material that might appear in the knowledge areas corresponding to these foundational concepts. In the current models, the knowledge areas for SE processes contain a lot of material that is there in its own right, rather than because it has a foundational role for the primary set of knowledge areas. Similarly, a knowledge area for software modelling would contain much material that was not specifically focused on unifying the areas across which this concept recurs. Hence, the material that is most directly concerned with structuring each of these dimensions should be factored out of these knowledge areas, and instead organised separately. This could be done either as a "foundational" knowledge area for each one, or else as knowledge units within one introductory knowledge area. The latter is probably preferable, since it would then have the explicit role of defining the structure of the rest of the model, and this should include the "people" dimension as well the ones for products and processes. Such a knowledge area would also provide an obvious location for some of the historical background that students need to be given, in order to set in context their study of the technical aspects of the discipline of SE.

8. Modelling in SE Programmes

Underlying these theoretical issues of the meta-modelling of knowledge are some highly practical ones, of what needs to be taught to students on an SE degree programme, and in what order. At the start of an undergraduate degree in SE, it can not be assumed that students will

know anything in particular about developing software systems, any more than it can be assumed they will know much basic CS. They may know something about programming, and if not they will need to be taught it, so that they can create algorithms and data structures as required. As well as this, the differences in scale between programs and software systems means that students must also master the structures that are used in creating complete systems. These structures are at the heart of software product modelling, and so this topic covers precisely the concepts that students need in order to understand software systems or create them, which is why it has to be taught right at the start of any undergraduate programme in SE.

Indeed, in the course of learning SE students effectively have to go through two main stages (or three, if one counts learning basic programming as a separate stage zero). Stage one covers the activity that one might call "software development", meaning the production of pieces of software that represent feasible solutions for some basic set of functional requirements. In terms of disciplines this activity lies in the intersection of SE and Information Systems, so that many of the textbooks to support it have titles that refer to systems analysis and design. They also often have a focus on the use of UML, which is now such a well-established notation that, whatever ones view may be of its technical merits, a graduate from a degree programme in SE must need some knowledge of its basic principles to be regarded as properly qualified. Starting off the teaching of SE with such an introduction to software development, particularly if it is strongly based on practical projects, will help to provide this, as well as giving the students a foundation for subsequently taking a more critical view of it.

By comparison with real SE, this topic of software development deliberately takes a very restricted view of many activities. Thus, it pays virtually no attention to issues of process, since typically the teaching has little alternative but to follow a basic waterfall sequence, and the same is likely to be true of any projects that students carry out as part of it. Similarly, virtually no attention is paid to issues of finding optimal solutions, rather than just feasible ones. This means that aspects such as product quality or process quality have to be confined to the practicalities of how basic validation and verification are performed, rather than attempting to quantify these concepts at a point in their education where students are still struggling with the binary problem of whether their systems work at all. Indeed, the benefit of this restricted approach is precisely that it abstracts away from (in other words, avoids the complication of) issues that the students are not really equipped to understand at this stage.

Once students have got within sight of the point where they can develop systems that will actually deliver some basic functionality, it is then realistic to go on to stage two, and actually teach them SE. Of course, it almost certainly is good to mention at least some of the distinctive issues of SE right at the start of stage one, just to set the context, but it is only at stage two that it is actually realistic to expect students to understand and use them. This is because engineering has to be based on theory, and software modelling provides a fundamental part of this, such as the ways in which models are created during requirements analysis, transformed during design, used to generate equivalents (ie code) during construction, and checked for consistency and completeness during validation and verification. Only once these have been properly mastered, as part of the software development stage, will the students be in a position to either appreciate why "real" SE (including its theory) is important, or to do it themselves.

9. Conclusions

The main conclusion of this paper is that software modelling, as described in section 2, is important to the SE curriculum on both theoretical and practical grounds. The theoretical ground is that the structures and properties of the models with which it is concerned capture the basic theory that underpins the whole of SE as an engineering discipline. The practical ground is that

students in the early stages of learning SE need to master the use of at least its basic models in order to carry out any of the activities of actually developing software systems, which is the essential purpose of SE.

Unfortunately, by comparison with the central role that modelling has in other engineering disciplines, the current curriculum models do not give this topic its proper emphasis. The main reason for this is the difficulty caused by its status as one of two foundational recurring concepts, the other being the concept of SE processes that has been used to derive the primary set of knowledge areas in each of these models. These two recurring concepts are orthogonal, and the solution that is proposed to the problem of documenting the resultant matrix structure as a linear hierarchy is to create an explicit knowledge area for software modelling, in the same way that ones have been created for SE processes. The elements of these concepts that actually provide the structure for the discipline should then form a separate introductory knowledge area.

Within SE programmes, the importance of modelling should then be reflected in a two-stage approach. Stage one, called software development, covers the basic models and their use in the core activities of SE. Stage two then uses this as a basis for teaching real engineering, drawing as necessary on the underlying theory provided by these models.

References

- 1 D J Bagert, T B Hilburn, G Hislop, M Lutz, M McCracken, S Mengel, *Guidelines for Software Engineering Education Version 1.0*, Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-99-TR-032, (August 1999), and also at <http://www.sei.cmu.edu/publications/documents/99.reports/99tr032/99tr032abstract.html>.
- 2 A B Tucker (ed), *Computing Curricula 1991*, ACM/IEEE Computer Society Joint Curriculum Task Force Report, ACM Press and IEEE Computer Society Press (1991).
- 3 G Booch, J Rumbaugh & I Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.
- 4 J M Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1989.
- 5 C B Jones, *Systematic Software Development Using VDM*, Prentice Hall, 1986.
- 6 J B Wordsworth, *Software Engineering with B*, Addison Wesley, 1996.
- 7 IEEE Computer Society, *Guide to the Software Engineering Body of Knowledge: A Strawman Version*, available at <http://www.lrgl.uqam.ca/publications/pdf/365.pdf>.
- 8 ISO/IEC, *Information Technology – Software Life Cycle Processes*, International Standard ISO/IEC 12207:1995(E), (1995).
- 9 IEEE Computer Society, *Guide to the Software Engineering Body of Knowledge: Trial Version (0.95)*, IEEE Computer Society (May 2001), and also at <http://www.swebok.org/stoneman/version095.html>.
- 10 IEEE-CS and ACM Joint Task Force on Computing Curricula, Approved Final Draft Version of Computing Curricula 2001 (15th December 2001), at <http://computer.org/education/cc2001/final/index.htm>.
- 11 A. E. K. Sobel (ed), *Computing Curricula – Software Engineering Volume: First Draft of the Software Engineering Education Knowledge (SEEK)*, 28th August 2002, available electronically at <http://sites.computer.org/ccse/artifacts/FirstDraft.pdf>.
- 12 A. J. Cowling, The first decade of an undergraduate degree programme in software engineering, *Annals of Software Engineering* 6, 61 – 90 (1998).
- 13 A. J. Cowling, What Should Graduating Software Engineers Be Able To Do?, these proceedings.
- 14 A Stratton, M Holcombe & P Croll, Improving the Quality of Software Engineering Courses Through University Based Industrial Projects, In M. Holcombe, A Stratton, S Fincher & G Griffiths, *Projects in the Computing Curriculum: Proceedings of the Project 98 Workshop, Sheffield 1998*, Springer Verlag, London, pp 47 – 69 (1998).
- 15 M Holcombe, M Gheorghe, F Macias, Teaching XP for Real: some initial observations and plans, In *Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 14 - 17, (2001).
- 16 B. W. Boehm and R. Ross, Theory W Software Project Management: Principles and Examples, *IEEE Transactions on Software Engineering* 15, 902-916 (July 1989).
- 17 A. J. Cowling, Structuring the Disciplines Related to Software Engineering: A General Model, In *Proceedings of the 14th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, 231-239 (2001).
- 18 A J Cowling, A Framework for Developing the Software Engineering Curriculum, *Proc. ACM/IEEE International Workshop on Software Engineering Education*, Sorrento, 1994, pp 111 - 118.