The role of modelling in the software engineering curriculum

A.J. Cowling

Department of Computer Science, University of Sheffield, Regent Court, Portobello Street, Sheffield S1 4DP, UK

Available online 12 May 2004

Abstract

This paper argues that the concept of modelling, and particularly the modelling 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; Curriculum structure

1. Introduction

In recent years there has been considerable interest in the issues of what software engineers need to know, and hence of what needs to be taught to students of software engineering (SE from now on). As work on this has progressed, through the projects to develop the SE body of knowledge (or SWEBOK) and then Computing Curricula 2001 (CC2001) and then Computing Curricula 2001 (CC2001 from now on), and particularly its SE volume (CC2001SE from now on), so it has become apparent that the major problems lie not so much in identifying the content as with determining how best it should be structured. A key issue here is to identify the concepts and relationships that underlie this structure, and the purpose of this paper is to discuss one concept which appears to be fundamental to this, namely that of modelling, and particularly the modelling of the products that SE is concerned with developing.

The background to this paper is that a previous version of it (Cowling, 2003a) had been concerned primarily with trying to remedy the apparent neglect of this concept, and hence with trying to influence the discussions within the SWEBOK and CC2001 projects. To some extent this goal was achieved (although there may well be more to be said towards it), and this version of the paper is being written in the context of the release of the first public draft for the full CC2001SE (IEEE and ACM, 2003). Hence, the focus has shifted, and so the aim of this paper is to assist those who must work with this document and its successors, by explaining not just the role of software product modelling, but also its relationships with other fundamental concepts. The significance of these relationships is that they give rise to a structure that is significantly more complex than the hierarchy of knowledge areas, knowledge units and topics that is now conventionally used for documenting curriculum models. Consequently, if SE educators are to work effectively with models of the SE curriculum that are documented in this conventional form, they need to understand both this more complex underlying structure and its effects, and it is these that this paper sets out to explain.

The starting point for the paper is therefore a description, in Section 2, of the scope and organisation of both software modelling as a whole, and software product modelling as the main part of it. Section 3 describes the role that this topic has had within SE, as
evidenced by various published sources describing aspects of the SE curriculum, and in particular the Guidelines for SE Education (Bagert et al., 1999), where it appears as a “recurring concept”. Section 4 compares this with the role that software modelling has had within the curriculum for one well-established undergraduate degree programme in SE, namely the one in the department of Computer Science at Sheffield University. Section 5 then contrasts these with the practice in other branches of engineering, where much more emphasis is put on the role of modelling. Section 6 returns to the idea of recurring concepts, and examines in more detail the significance of treating software modelling as one of these, and Section 7 then goes on to discuss the relationship of this with the conventional hierarchical structure used to document curriculum models. To bring out the practical relevance of what would otherwise be a fairly theoretical discussion, Section 8 discusses 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 above, 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 (Tucker, 1991—referred to as CC91 from now on). One of these seven recurring concepts is entitled “Software Modelling”, and the definition for it that is given within the SE Guidelines is as follows. “The Software Modelling 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 Fig. 1.

At the top level of this hierarchy there are two aspects of modelling that are to some extent orthogonal, but which both need to be covered within the SE curriculum. One of these aspects is concerned with the types of models that can be built, in terms of how “abstraction, modularity and hierarchy” can be applied, and this in turn divides into two. One of these two types of model at the second level of the hierarchy is the qualitative or structural model, which is concerned with describing the different kinds of concepts (such as modules, functions or “data objects”) that occur in the structure being modelled, and the various relationships between them (such as “hierarchy”) that go to make up this structure. The other type of model at this second level is the quantitative model, which describes the measurable properties or attributes of the objects being modelled, and the relationships between these attributes. These models will therefore usually be expressed in terms of equations that involve variables representing the values of these different attributes, so that the equations define the relationships between these attribute values, and their forms will usually be derived in part from the underlying structures that are described in the qualitative models of the objects in question.

By contrast, these qualitative models can take either of two forms, and these are identified as separate elements in Fig. 1. One group consists of the diagrammatic models, which typically use boxes for concepts and lines for relationships, as in UML (Booch et al., 1999) and earlier notations such as SSADM (Goodland and Slater, 1995). The other group consists of formal models, which are based on similar uses of appropriate mathematical constructions, and are typically expressed within notational frameworks such as Z (Spivey, 1989), VDM (Jones, 1986) or B (Wordsworth, 1996).

The other main aspect of modelling identified at the first level of the hierarchy is concerned with the purposes for which models are used, meaning the different kinds of structures (such as “software architectures” and “development entities”) or properties that may need to be modelled. This aspect again divides into two, since one purpose is to build and use models of SE products or systems, and the other (which is not explicitly mentioned in the Guidelines definition) is to model SE processes. The product aspect can then be broken down into three main areas: models for the functionality of systems (where the term is used here to also include the underlying “data object relationships”), models for their quality (which partly relates to the aspect of “behavior”), and models for the cost of construction.

Here, the models of system functionality are essentially qualitative, but there are several different approaches to how they should be classified, and so no attempt has been made to show any such breakdown in Fig. 1. Originally the structured analysis and design methodologies identified just three components,
shown in Fig. 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 components, in which each of the data and processing aspects are divided into a static part and a dynamic (behavioural) part, as shown in Fig. 2(b). In both halves of Fig. 2, the significance of the outer rectangles is that they reflect an important insight from formal methods, which is that these different components (data, processing, etc.) are actually not separate models, but rather different views that focus on particular aspects of what is actually one integrated model.

Fig. 1 also does not attempt to show the enormous variety of formal and diagrammatic models that exist, again because there is no established basis for classifying them. For instance, there have been several reviews of formal specification methods, notably by Gaudel (1994) and by van Lamsweerde (2000), but each proposes a different classification of the underlying models, and these structures have only a few elements in common, such as distinguishing algebraic approaches or state-based approaches from the rest. Similarly, there have been various reviews of the enormous variety of methods and techniques that utilise diagrammatic models, and a good account can be found in Wieringa (1998), which illustrates clearly that there are far too many of them to try to show any classification in this figure.

By contrast with the models of functionality, those for the quality and cost of construction of systems are essentially quantitative, and like the quantitative models of structure they are based on the models for the functionality of a system, in that their structure is derived essentially from the set of components and relationships that are identified within these qualitative models. A similar situation also arises with the models of SE processes, which at their most basic just represent a development process as being composed of a set of activities which are performed in some sequence, where the sequence may be described in terms of analogies such as a waterfall or a spiral. Such a breakdown of a process into activities then leads to the construction of models for the resources to be used within a process, and the time that each step in the process will take. Alongside this there are other aspects of a process that need to be modelled, such as the risks that can arise during it and the activities that can be undertaken to manage risk. Finally, the relationships between these different components lead to models for the quality of a process itself, as distinct from models for the quality of the artefacts produced by instantiating that process.

3. The current role of modelling in SE

Given this definition of the scope of software modelling, there is a significant contrast to be drawn between the extent to which it features in current sources and the role that it actually needs to play in delivering an SE curriculum. To demonstrate the first part of this contrast, the current sources that need to be considered can be split into two groups, where one group consists of the common introductory textbooks on SE, and the other group consists of the various documents that attempt to define models for the SE curriculum.

For the common introductory textbooks, a good estimate of the importance which they attach to modelling can be gained by checking their index and contents pages for references to it, in much the same way as was done in the early stages of the SWEBOK project (IEEE, 1998). This has been done for a reasonably large sample (although not quite as large as the one used for the SWEBOK), and 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 specifically 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 a 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. As noted above, the SWEBOK project started from a comprehensive analysis of the structures used in a set of the popular textbooks on SE, together with the ISO/IEC standard 12207 (1995), and since the analysis given above followed the approach of the one that they had made, it is hardly surprising that the two sets of results are similar. Thus, the Strawman version of the guide to the SWEBOK made little reference to modelling, and this is still the case with the current version (IEEE, 2001). 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 level of 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 CC2001 (IEEE and ACM, 2001) follows much the same pattern, since the authors of it were drawing on the same set of sources, together with the Guidelines document (which had been published some time after the start of the SWEBOK project). Here, an important constraint was the need to reduce the material so as to fit within a single knowledge area, with the result that many topics which one might have wanted to see included had to be condensed or even omitted. Thus, given the sources on which it is based, it is hardly surprising that modelling does not appear in the titles of any of the 12 knowledge units that make up this knowledge area.

On the other hand, modelling does appear in some of the topics that make up these units. Specifically, “Requirements analysis and design modelling tools” is one of the five topics in the unit “Software tools and environments” (SE3), two of the three topics in the unit “Software processes” (SE4) refer to modelling, “Requirements analysis modelling techniques” is one of the five topics in the unit “Software requirements and specifications” (SE5), and “Software reliability models” is one of the four topics in the unit “Software reliability” (SE11). Furthermore, there are a number of references to modelling in the descriptions of some of the units. Even so, the subject still has much less importance than either activities such as requirements analysis or design, or others of the recurring concepts from the Guidelines document. For instance, both “software processes” and “tools and environments” appear as knowledge units in their own right (respectively SE4 and SE3), possibly reflecting the fact that they are also (with slight variations in the titles) the subjects of chapters in the SWEBOK guide. Then, two more of these recurring concepts, “software quality” (which has a chapter to itself on the SWEBOK guide) and “software metrics” (which forms part of the chapter on SE management), are both represented strongly in the units on “software project management” (SE8) and “Software reliability” (SE11) respectively.

More recently, the first stage in developing CC2001SE has involved defining a model known as SE education knowledge (SEEK), which 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. At the time of writing this had been through three major versions, known respectively as the first (Sobel, 2002a), second (Sobel, 2002b) and final (Sobel, 2003) drafts, with the final draft being the one that is incorporated in the first draft of the full CC2001SE volume, so that it is only likely to change if this is needed to match changes in the rest of the volume. Of these three drafts, the first followed a very similar structure to the SWEBOK, and indeed it explicitly acknowledges its dependence on this as a key input. Consequently, its coverage of modelling is very similar, although there are more references to the concept, and this pattern continued in the second draft as well. Specifically, in both of these drafts “Modelling” is one of the units in the “Fundamentals” area, which consisted of four units in the first draft and five in the second; and “Requirements modelling and analysis” is one of six units in the “Requirements” area. Also, 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 unit which is called “User interface design” in the first draft and “Human computer interface design” in the second draft; “Modelling and specification of software processes” as a topic in the “Process concepts” unit; “Life cycle models” as a topic in the “Process implementation” unit; and “Software quality Models and metrics” as a topic in the “Software quality processes” unit.

While this coverage was more extensive than in SWEBOK, it did 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. To some extent this was rectified in the final draft, though, where one of the main structural changes was the replacement of the previous “Requirements” knowledge area by one entitled “Software modelling and analysis”. This area pulls together into a more coherent structure the units related to modelling that were previously in the “Foundations” and “Requirements” areas, although it combines them with other units from the previous “Requirements” area, such as “Eliciting requirements” and “Requirements specification and documentation”. For these other units the relationship to modelling is no stronger than it is for any
other aspect of SE that is represented by separate knowledge areas, such as “Software design” or “Software verification and validation”. Thus, while this final version of the SEEK does give much greater prominence to software modelling, the way in which it links it with requirements analysis could give the impression that modelling has less relevance to the rest of the activities within the SE process. This is not the case, though, as the rest of this paper will proceed to demonstrate.

4. Modelling and the Sheffield SE programme

Of course, this comparatively limited emphasis on modelling in these various sources would not matter if the subject itself were less important than some of these other topics, such as software processes, software measurement, software quality or software tools. The experience of the author and colleagues is, however, that this is not the case, and that modelling does need to be given at least as much attention within the SE curriculum as these other topics. This experience has been gained from running an undergraduate degree programme in SE in the department of Computer Science at Sheffield University, which started in 1988, so that it has now had a considerable time over which to develop. Much of this development has been described elsewhere (Cowling, 1998), and a feature throughout it has been the way in which it has been necessary at various stages to review the curriculum, and in particular to ask the questions “what are its goals, and is it achieving them?”. In general terms, the answer to the first of these questions has been essentially “to equip graduates to begin to function as software engineers”, and in trying to make this more precise the emphasis has been increasingly on the role of the skills that the students should develop within the programme, as highlighted in section E.1 of the SE Guidelines, and as discussed by the author elsewhere (Cowling, 2003b).

Here, the most important overall skill is that of being able to develop (or participate in developing) software systems, and a key component in trying to achieve this is the use of projects that involve the development of actual software systems, particularly if these can be real systems for real clients (Stratton et al., 1998). Reflecting this, the curriculum for this programme has recently undergone its fifth major revision, and an important principle that guided this revision is that the “spine” of the curriculum should be formed by a sequence of major projects, one in each academic year. Then, the content of the rest of the curriculum has been driven substantially by the need to ensure that the students have acquired the knowledge that they need for undertaking these projects.

For the purposes of this paper, it is the major project that takes place in the first year that is the most important, since this is where the foundation is laid for this ability to develop software systems. (As an aside, it should be noted that this reference to the first year is within the context of the three year pattern that is normal for a bachelor’s degree programme in the UK: in four year programmes such as those in the USA, where the first year provides much more of a general educational introduction, it would be more typical for the second year to play this role in the curriculum.) This project is carried out in teams, and it uses 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, as follows:

- in the requirements analysis stage, a conceptual model (mainly UML use cases and a class diagram);
- in the formal specification stage, a formal model (in Z);
- in the design stage, a design model (mainly UML sequence diagrams);
- in the implementation stage, the code itself (in Java); and
- in the testing stage, 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 appeared in the “Fundamentals” area of the first and second drafts of the SEEK, and were then distributed between the “Computing Essentials” and the “Mathematical and Engineering Fundamentals” areas in the final draft of it.

By contrast with this emphasis on model-based content, our experience has also been that relatively little teaching time needs to be spent in the first year on the process within which these activities are carried out, since once the basic concepts of processes, activities and deliverables have been covered, the students essentially learn the rest that they need for this project by doing it.
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. While it has given the students valuable insights into the contributions that tools can make to software development, 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 of the degree programme, however, the emphasis does have to switch more towards processes, since the projects which are carried out in the second year use external clients. This means that they are much more varied, and so the issue of how to adapt the development 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 (Holcombe et al., 2001). Even so, the more general concept of modelling still forms a significant element in the second and third years of the curriculum, underpinning as it does 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 10 out of the 12 texts examined 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. Moreover, 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, so that neither of them could be regarded as representative. The rest cover a set of models that range from stress and strain in soils, constructions and machinery; through models of corrosion, fatigue and creep in materials, and on through equations for mass transfer and for material and energy balance in reactions; to models of dynamic system responses, signal representations and transfer functions; and finally to models of electrical and electronic circuits.

The conclusions of this research 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 the “Theory W” of Boehm and Ross (1989), which has only very limited relevance to the issues being discussed here.

6. Modelling as a recurring concept

This clearly confirms the argument that the concept of modelling ought to play a more important role in the SE curriculum than it currently does, and particularly the modelling of software system structures. This, however, creates a problem for those trying to develop models of the SE curriculum itself, 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, since it has been demonstrated elsewhere (Cowling, 2001) 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 12 concepts, and the analysis suggests that these can be classified into four categories, depending on why they possessed these characteristics and the extent to which they abstract from the topics over which they recur. These four categories are summarised in Table 1, and they are defined as follows, where the examples are taken both from CC91 and from the SE Guidelines.

The most abstract of these categories consists of the “emergent” concepts, meaning those that are important primarily because they recur through describing emergent properties of other concepts. Here, “emergent” is used in the sense that these properties appear in different forms in the various knowledge areas, where the variations in form are sufficiently significant that this is as far as one could go in describing these concepts as recurring. A number of the concepts from CC91 come in this category, such as those listed in Table 1, but none of the recurring concepts from the SE guidelines exhibit such a variety of forms. Emergent properties like this are important because they help to integrate different parts of a curriculum, but this effect primarily contributes to the delivery of the material in the curriculum rather than to its actual structure. Thus, while the absence of such recurring concepts from the CC2001 models is regrettable, it reflects this fact that they are not actually a structural feature.

The next most abstract of these categories consists of the “applicable” concepts, meaning ones that recur because they describe features that do have a consistent form, but that are important primarily because they can be applied to the material in various knowledge areas. The obvious examples here from CC91 are those listed in Table 1, all of which constitute issues that need to be considered in their own right in the course of developing software systems. The same characteristics are also possessed by several of the recurring topics in the SE Guidelines, so that at each stage in developing a software system a software engineer will need to consider how to apply each of these concepts to this development work.

In terms of the structure of a curriculum model, the effect of a concept recurring for this reason is that there will need to be some basic material somewhere in the curriculum to define the concept itself, but the main focus of its treatment will need to be the material that relates to the way in which it is applied within each of the knowledge areas where it recurs.

The third category of recurring concepts consists of those that are “structural”, meaning that they not only apply to the material in various knowledge areas, but are sufficiently closely linked to it that they also provide some basis for structuring it. For instance, in CC91 the concepts of “binding”, of “levels of abstraction” and of “tradeoffs and consequences” come in this category, because in each case they arise from the methods of analysis used in the topics where they recur, and hence

<table>
<thead>
<tr>
<th>Table 1 Proposed categories of recurring concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category and examples</strong></td>
</tr>
<tr>
<td>Emergent</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Applicable</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Structural</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Foundational</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
help to structure the treatment of these topics. Similarly, in the SE guidelines the topics of “software quality” and “software metrics” certainly come into this category, because for each of the main knowledge areas (which were defined in terms of the stages in the software lifecycle) they arise from the very structure of the activities described in that area, and so there needs to be a distinct component of material that is concerned with how this recurring concept impacts on these activities. Of course, as with the “applicable” concepts there also needs to be some basic material defining the concept itself, but the difference here is that the concept is so significant that it actually shapes the treatment of some of the material to which it is applied, rather than just being related to it.

The final category of recurring concepts consists of those that are “foundational”, meaning that they recur because they actually underpin the very basics of the material in other knowledge areas. There is only one recurring concept from CC91 that seems to fit this definition, and this is “conceptual and formal models”. As such it is the precursor of the “software modelling” concept in the Guidelines, which therefore also needs to come in this category, because (as has already been argued) the notions of creating and transforming models are so fundamental to any of the stages in the SE lifecycle that it is almost impossible to describe them sensibly without referring to this recurring concept. Of course, this does mean that the concept of “software processes” from the Guidelines ought to come in this category too, because it includes the “definition... of software engineering processes”, and as such it is fundamental to the whole basis on which the main knowledge areas within the Guidelines were identified.

This classification of recurring concepts implies that the foundational ones should have nearly as important a role to play in structuring a curriculum model as the basic knowledge areas do, and this is certainly what has happened with “software processes”, which forms a knowledge area (or equivalent) in each of the SWE-BOK, CC2001CS and SEEK models. The question that therefore needs to be asked is why the other foundational recurring concept, namely “software modelling”, has not been given anything like the same importance. To some extent the answer to this has to be conjecture, but the key seems to lie in the fact that a number of the main knowledge areas (meaning those relating to different stages in the software lifecycle) can be derived from the concept of software processes, so that the latter not only recurs across the former but also obviously supports them. By contrast, while the concept of software modelling recurs across these knowledge areas, it is orthogonal to them, and this aspect of orthogonality complicates significantly the underlying structure of curriculum models. It is therefore not surprising that attempts have been made to try to avoid it, but unfortunately this complication is inherent in the nature of the subject, and so it cannot just be avoided, but instead must be managed.

7. Modelling and curriculum structures

The nature of the complication that is caused by this property of orthogonality can be stated very simply. If there are two sets of orthogonal features that must both be included, then the underlying structure is that of a two-dimensional matrix, and this cannot readily be documented as a one-dimensional list or set, which is the most convenient structure for human readers to follow. Thus, in this case, the orthogonal nature of the two concepts of software modelling and software processes means that the SE curriculum really needs to be visualised as a matrix, in which one dimension corresponds to the concept of software processes and the other dimension to the concept of software modelling. In such a matrix, the software process dimension will be structured into components corresponding to the usual set of activities within the process, from requirements analysis through to construction. The structure of the modelling dimension will be derived from the one illustrated in Fig. 1, and so will have as its elements the terminal nodes of this structure, apart from the node for process models, since this will form the basis of the other dimension. The elements of this matrix will then correspond to the ways in which the relevant models are used in the various process activities, such as being constructed initially, being transformed, being checked for consistency with other models, etc. Hence, the basic structure of this matrix will be as shown in Fig. 3.

If the curriculum structure is viewed in this way, then most of the recurring concepts from the SE Guidelines also fit naturally into it, since tools and environments, software quality, software metrics and documentation can all be seen as being at least partly derived in some way from either the properties that are being modelled or aspects of the handling and presentation of models. Indeed, the only one of these recurring concepts that does not fit in at all in this way is ethics and professionalism, although there are also significant aspects of the concepts of tools and environments and of documentation that go beyond the features represented in

---

Fig. 3. The matrix structure of the curriculum.

---
this matrix. The explanation of this is that actually the matrix should be seen as having three dimensions rather than just two, with the three dimensions corresponding to products, processes and people (as identified originally in Cowling, 1994), so that the concept of ethics and professionalism belongs to the structure of this third dimension, namely that of people. The same is also partly true of the other recurring concepts, in that they have both social and technical aspects, where the distinction between the two forms part of the structure of this “people” dimension, and then the technical aspects fit into the matrix structure shown in Fig. 3.

In any subject area where the underlying material possesses such an inherent matrix structure, then if the idea of recurring concepts is not going to be used explicitly to represent it, the general problem of creating a satisfactory one-dimensional curriculum model has two parts. One part is to flatten this matrix structure into a linear hierarchy, with the conventional levels of knowledge areas, knowledge units and topics. The other part, which is the more difficult problem, is to do this while also retaining and making explicit the essential relationships between the components of this hierarchy that derive from the underlying matrix structure.

For the first part the obvious approach is to select one of the dimensions of the matrix, and to take the structure of this as the basis for a primary set of knowledge areas. This is then augmented by a secondary set of areas, which represent those components from the other dimensions that provide material which would not be covered adequately by the primary set. Thus, the secondary set will contain at least some of the orthogonal (i.e. recurring) concepts, and so if the second part of the problem is to be solved, then at the very least their structure (in terms of knowledge units and topics) has to reflect to some extent the relationships that result from this orthogonality. This is why the problem of finding the best structure for the knowledge areas and knowledge units in the SE curriculum has proved such an awkward one, since almost any solution to this second part of the problem is likely to be incomplete in some respects.

Looking at the general problem of flattening the matrix structure of the SE curriculum, then in principle one could try to solve it by selecting any one of the three dimensions as the primary one, and using it as the basis for a curriculum model. In practice, though, the “people” dimension essentially just has three main components, where (as indicated above) the first level of structuring identifies technical issues and social issues, and then the second level divides social issues into those relating to the roles of people as users of software systems and those relating to their roles as developers of them. Consequently, this dimension does not have sufficient components to provide a good starting point for such a flattening process, and so this just leaves the “product” and “process” dimensions as realistic choices. Furthermore, all the models referred to above have followed each other in selecting the “process” dimension as the primary one, so that effectively they have started by identifying a knowledge area for each column of the matrix. Thus, while it might be interesting to explore what sort of model would be obtained if it were based on the “product” dimension (i.e. starting with a knowledge area for each row of the matrix), the choice of the “process” dimension as the primary one will be used as the basis for the rest of this discussion.

Thus, in any of the recent efforts to develop curriculum models for SE, there have been two key issues that have needed to be solved. One, corresponding to the first part of the structuring problem, has been how the components from the “product” dimension should be used to augment the primary set of knowledge areas. And, of course, the “people” dimension should in principle be included in this too, but in practice this has just meant including a knowledge area for the various social issues, and the structure of this is fairly self-contained, and hence well understood. The other key issue, which corresponds to the second part of the structuring problem, has been how to ensure that the resultant model adequately represents the orthogonality relationships between the product and process dimensions.

For the first of these issues there is a basic set of additional or secondary knowledge areas on which the different models are agreed. All of them include in their secondary sets knowledge areas for software project management (which for the Guidelines was actually in the main set) and for software quality (although in CC2001CS this is specialised to software reliability). Three of them include software tools and environments, with the exception being the SEEK, which treats these topics instead as a knowledge unit in the area that in the final draft is called “Computing Essentials”. A couple of other topics appear in several of the models, but at different levels in the hierarchy. One of these is software configuration management, which is a separate chapter (i.e. knowledge area) in the SWEBOK model, but forms part of software project management in the other models. The other such topic is software metrics, which forms a recurring concept in the Guidelines, but is spread across several other areas in both the SWEBOK and the SEEK.

By contrast, the issue that has not been fully solved in any of these models is that of representing properly the relationships between the primary and secondary sets of knowledge areas, in order to make clear that they are derived from the underlying matrix structure. Here there is a useful analogy that can be drawn with two fundamental topics in CS that are also orthogonal, namely data structures and algorithms. This analogy is based on the observation that software product models essentially
describe the structures of the data about a software system that is created and used in the activities that make up SE processes, while these different activities then describe (in a fashion that one would like to regard as more or less algorithmic) how this data is processed in order to produce that system. In CC2001CS these orthogonality relationships between data structures and algorithms are handled by structuring the material into two parts, where the first part defines the foundational concepts of each as separate knowledge areas in their own right, namely “Discrete Structures” and “Programming Fundamentals”, and then the second part organises the material where they inter-relate into a separate area, “Algorithms and Complexity”.

Scaling this two-part approach up to software product models and SE processes, the effect would be that the foundational concepts of each should form a separate knowledge area, or possibly several areas. Then, the material where they inter-relate would effectively be what has been called here the primary set of knowledge areas. Currently the existing models go part way towards using such an approach, in that they all include a knowledge area for SE processes, reflecting the fact that it is a foundational recurring concept. Consequently, their primary knowledge areas have largely omitted material that just relates to the process dimension on its own, and so to this extent this approach has enabled these areas to focus on the relationships between products and processes.

As yet, though, none of the current curriculum models have gone as far as including an equivalent knowledge area for software modelling, as the other foundational recurring concept, although the final draft of the SEEK gets nearer to this, with its knowledge area that links modelling and requirements analysis. This linkage is understandable, since (as Fig. 3 indicates) it is during the requirements analysis activity that many of these models are constructed initially, but it still has the effect of implying that modelling is only relevant to requirements analysis, when in fact it underpins all of the process activities that form the other dimension of the matrix. Consequently, there are still some foundational aspects of modelling (such as design notations) that appear in the knowledge areas for these other process activities, rather than being separated out from them.

Having separate knowledge areas just for the two foundational recurring concepts of modelling and processes would therefore go quite a long way towards solving the problem of representing the orthogonality relationship between products and processes, but there is still more that could be done in order to make this relationship explicit. In particular, in the various proposals for the content of a knowledge area for SE processes there is a considerable amount of material that is there in its own right, rather than because it has a foundational role on unifying the primary set of knowledge areas. Similarly, a knowledge area for software modelling would almost certainly contain much material that was not specifically focused on unifying the secondary set of knowledge areas.

Thus, it might also help to make the relationships between the product and process dimensions of the curriculum more explicit if the material that was most directly concerned with structuring each of these dimensions were factored out of these foundational knowledge areas, and instead organised separately. This would require the creation of some kind of introductory knowledge area, which would have the explicit role of defining the structure of the rest of the model in terms of grouping the other knowledge areas into foundational, primary and secondary sets. Such an introductory knowledge area would thus consist primarily of a unit for each of the three dimensions: one each for products and processes, and in practice it would be sensible to include one for the “people” dimension as well. This area might also include one or more units for some of the historical background that students need to be given in order to set the discipline of SE in context, such as the relationships and distinctions between science and engineering, and the role of the software crisis in the development of SE.

This is where the issue of how the curriculum model should be structured starts to give way to the practical considerations of how it should be delivered within SE programmes. Consequently, from a discussion that so far has been phrased largely in theoretical terms, almost at the level of meta-modelling of knowledge, attention needs to turn to the highly practical issues of what needs to be taught to students on an SE degree programme, and in what order it needs to be taught.

8. Modelling in SE programmes

At the start of an undergraduate degree in SE, it cannot be assumed that students will have any particular knowledge of how to develop software systems, any more than it can be assumed they will have much knowledge of basic CS. They may well know something about programming, and if not they will need to be taught it, so that they will be capable of creating algorithms and data structures as required. This is not enough, though, for the differences in scale between programs and software systems are such that the latter simply can not be understood just in terms of these basic programming concepts. Rather, understanding them requires a familiarity with the more elaborate structures that are used in creating complete systems, and it is these structures that are at the heart of software product modelling. Consequently, this topic covers precisely the concepts that students need to be able to understand
software systems or create them, and this is why it needs 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 at least two main stages, and probably three, since it seems appropriate to count learning basic programming as a separate stage zero, that covers much of the material in the “Computer science foundations” unit of the SEEK. Then, stage one is to learn about an activity that will be called here “software development”, and that forms a precursor to actually teaching SE properly in stage two. The significance of describing this stage one activity as software development is that it involves the production of pieces of software that will represent feasible solutions to the problem of meeting some basic set of functional requirements. In terms of disciplines this activity lies in the intersection of SE and Information Systems, and it is supported by many textbooks with titles that refer to systems analysis and design, which typically these days focus on the use of UML within these activities. Despite this, though, the argument that this topic needs to be taught as a precursor to SE is not intended as any kind of comment on the various technical issues relating to UML, or on its relationship to other modelling notations. On the other hand the influence of UML cannot be ignored, to the extent that it is difficult to see how a graduate from a degree programme in SE could be regarded as properly qualified unless they had at least some basic knowledge of the principles underlying it. Starting off the teaching of SE with an introduction to the ways in which it is commonly used would seem to be the best way of ensuring that students understand what UML can contribute to SE, as well as laying a foundation for them to be able subsequently to appreciate a more critical view of it and its role.

Thus, the teaching of this activity of software development has to be strongly practical, which is why the principle that practical projects form the spine of the curriculum has been found to be so useful at Sheffield University. The other feature of software development that is important to its role as a precursor to SE proper is that, quite deliberately, it takes a very restricted view of what is involved in developing software systems. Thus, within this activity virtually no attention is paid to issues of process, since typically any projects that students undertake as part of the teaching of software development will have little alternative but to follow the basic waterfall sequence, and so the same is likely to be true of the teaching itself. Similarly, within this activity virtually no attention is paid to quality or cost requirements, except for the very crudest expressions of them, such as “the system should be user friendly”, which for this purpose can be taken as meaning in practice that systems to be developed or studied should have graphical user interfaces rather than command-driven ones. Consequently, no attention is paid either to the issues that would arise from considering such requirements, namely those of trying to find good, or even optimal, solutions to the functional requirements, rather than just feasible ones. Hence, at the stage of teaching software development to students, issues of 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.

The benefit of this restricted approach to software development is 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. Only once they have got within sight of the point where they can develop systems that can actually deliver some basic functionality is it realistic to go on to stage two, and actually teach them SE as opposed to just software development. 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 teach students about them and expect them to understand and use them.

These issues that are distinctive of engineering include: how requirements can be clarified and quantified (particularly in the face of uncertainty or changing contexts); how systems can be designed to best meet given combinations of quantitative requirements; how particular design techniques can improve important characteristics of systems; how risks can arise and be managed within development processes; and how changes to versions and configurations of systems can be managed. And, of course, because this is engineering, they must also include the theory that underlies all these practical issues. In turn, this theory is based on the fundamental concepts of software modelling, and in particular the ways in which models need to be created during requirements analysis, transformed and evaluated quantitatively during design, used to generate equivalents (i.e. code) during construction, and checked for consistency and completeness during validation and verification. Consequently, it is only when these model-based concepts within SE have been properly covered, as part of the software development stage, that students will be in a position either to appreciate why “real” SE (including its theory) is important, or to do it themselves.

9. Conclusions

The conclusions that can be drawn from this cover three aspects: the status of modelling itself, its
theoretical impact on models for the SE curriculum, and its practical impact on degree programmes in SE. For the first of these, the role of modelling within the whole discipline of SE, the conclusion has to be that it is not currently given enough attention, apart from one or two specialised areas, such as process modelling. In particular, not enough attention is given to modelling the structures of the products of SE, namely software systems, despite the fact that the creation and manipulation of such models forms the very foundation for the core activities of SE. Hence, the conclusion also has to be that the study of these models needs to be given more emphasis, both to provide this foundation and to bring practice in SE education closer into line with practice in other branches of engineering.

To reflect this increased emphasis within the models for the SE curriculum, the conclusion is that two steps are required. The first step is that software modelling needs to be recognised as an essential area of knowledge within these models, whether or not this is actually structured as a formal “knowledge area”, and where the content of this area of knowledge would be roughly as outlined in Section 2 of this paper. Ideally this knowledge area should have equal status in the formal structure with the one that describes SE processes, but the important thing is that the foundational role of these two knowledge areas needs to be recognised. The second step is then to address the problem created by the orthogonal dimensions of the curriculum structure, so as to distinguish those topics that involve the relationship between software modelling and process activities from those that just relate to the models or the activities themselves.

Part of the solution that has been suggested to this second step is to have the foundational knowledge areas separate from those covering topics that involve their intersection. The other part of this suggested solution is the identification of a separate introductory knowledge area, which would have the explicit role of describing the three dimensions of the original matrix model. That is, the role of each of the main units in this area would be to define the structure of one of these dimensions, and the effect of this would be to group the other knowledge areas into a primary set (corresponding to the principal dimension, namely that of SE processes) and a secondary set (corresponding to the recurring concepts, namely those making up the product modelling dimension).

At the practical level, the impact that this increased emphasis on modelling would have on the structure of undergraduate degree programmes in SE would be to provide a more structured way of building up the coverage of one of the core skills that software engineering students need to acquire, namely the ability to develop software systems. This more structured approach has three stages, although the first of these (stage zero) is simply that of learning fundamental programming skills and other related computer science concepts. Then, stage one consists of what has been called here software development, and stage two is the one that actually consists of teaching SE.

The difference between these is that stage one is concerned with just developing systems to meet basic functional requirements, so that this stage is only concerned with qualitative software product models. Then, stage two goes on to consider aspects such as how quantitative requirements can be met, or how to develop systems that are closer to being optimal solutions to problems, rather than just feasible ones. This stage will, of course, also require work that is based on product models, but it will also rely heavily on process models and on the impact of social aspects of systems, and hence the role of modelling will be much more broadly based, and in particular will be as much quantitative as qualitative. As such, this two-stage approach to the teaching of SE proper will be much closer in spirit to the approaches used in other branches of engineering, which must surely be a desirable goal in the current state of development of SE.

References


Tony Cowling obtained his BSc in Computer Science and Mathematics from the University of Leeds, and also his PhD in Computer Science. Since 1973 he has been first a lecturer in Computer Science at the University of Sheffield and more recently a senior lecturer. He became director of undergraduate studies in the department in 1988, overseeing the introduction of its Software Engineering programme, and is now director of teaching quality for the department. He is a member of the British Computer Society, a Chartered Engineer in the UK, and a European Engineer (Eur Ing). His research and teaching interests are all in the area of Software Engineering, and specifically in formal modelling and software testing.