Stages in Teaching Software Testing

Tony Cowling
Department of Computer Science,
University of Sheffield,
Sheffield, United Kingdom
A.Cowling @ dcs.shef.ac.uk

***Abstract*—This paper describes how a staged approach to the development of students' abilities to engineer software systems applies to the specific issue of teaching software testing. It evaluates the courses relating to software testing in the Software Engineering volume of Computing Curriculum 2001 against a theoretical model that has been developed from a well-established programme in software engineering, from the perspectives of how well the courses support the progressive development of both students' knowledge of software testing and their ability to test software systems. It is shown that this progressive development is not well supported, and that to improve this some software testing material should be taught earlier than recommended.**

***Keywords*-software engineering; software education; software development; development of skills**

I. INTRODUCTION

Software testing is not a glamorous topic within software engineering (SE from now on), and particularly not when compared with such major ones as requirements analysis or software design, but it is still an important topic for SE students to master. This is because the main goal for such students is to develop the skills needed for engineering software systems, and one of these skills is that of ensuring that, once they have been engineered, such systems do actually meet the requirements that were originally set for them. This is the skill of testing the system against its requirement, and it is of course a complex skill, which involves applying knowledge of a number of different areas within SE. Hence, it can not all be learnt in a single step, but rather the component skills involved in software testing need to be built up in stages. This structure of stages in building up these skills is the primary concern of this paper.

The key part of the background to the paper is that there are several different structures for such stages defined in the SE volume of Computing Curricula 2001 [1] (SE2004 from now on). These course structures are defined in chapter 6, which describes various alternatives for the introductory and core sequences of SE courses, and these have different treatments of software testing. Also, a lot of basic material on testing is classed as Computer Science (CS from now on), and so this material is imported into the SE2004 model from the equivalent volume for CS [2] (CS2001 from now on). In this imported model testing is broken down into a number of components, and various alternative courses are then defined to cover these components, each describing a different progression path through them.

To some extent the different course structures in SE2004 reflect the different approaches of CS2001, although they are driven more by the problems of fitting CS and SE material together and then packing it into course-sized chunks. As a result, for specific skills (such as software testing) the progressive development is not always defined within these course sequences as clearly as perhaps it needs to be. This is particularly significant because one of the main emphases of the guiding principles for the whole volume (described in its chapter 3) is on student-centered learning outcomes, which are discussed in its section 3.2. All of these outcomes are focused on skills, and since they are supposed to underpin the structures of the course sequences, it is important to analyze just how well these structures do actually support the students in developing the relevant skills.

The purpose of this paper is therefore to evaluate these course sequences, from the specific perspective of how well they support the progressive development of software testing skills. An important part of the context for this evaluation is the recognition that the processes of defining the overall structures of these courses and of defining the details of individual courses were in practice slightly different, even though this is not made explicit in SE2004 itself. Even though these differences were not substantial, they do imply that the details of individual courses may have to be treated as less authoritative than the overall structures.

Specifically, the overall course structures, such as the division into introductory, core and additional courses, and the approach to reusing introductory courses from CS2001, were developed by the pedagogy focus group for SE2004, as described in section 1.3.2 of the volume. The details of each particular course were then defined by quite small groups of individuals, and the reference in this section of the volume to this being done "using existing programs as a guide" actually means that in many cases those individuals were simply incorporating details of courses that they had already developed and taught. Furthermore, the only public reviews of the course structures were as part of the reviews of the volume as a whole, where typically the reviewers (of whom the author was one) were sufficiently experienced SE educators that their natural inclination would have been to treat the details as a rough guide rather than as a precise specification. Consequently, they did not critique these details as rigorously as they did other components of the

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland
Software Engineering Education

volume, as indicated by the fact that almost none of the published review comments [3, 4] suggested amending the detailed mappings of topics into courses.

Indeed, the whole issue of the role of course structures in model curricula is one where expectations appear to be changing. In the days when institutions were likely to create a degree programme by taking an appropriate model curriculum and instantiating it almost "off-the-shelf" – a situation which the author has seen in quality reviews of some degree programmes outside the UK – then the detailed course descriptions were extremely important elements of the models. Now, however, model curricula are more usually used as guides for reviewing or adapting existing degree programmes, rather than as templates for creating new ones, and so the underlying structures become more important than the details. This is why the planning for the new version of the CS model curriculum (known as CS2013) envisages that detailed course descriptions should be treated as exemplars, rather than as actual components of the model [5]. This shift in the role of course descriptions makes it all the more important to emphasize the principles that should be used to determine the structures and content of individual courses, and so the main goal of this paper is to explain some of these principles and show how they impact on the content of individual courses.

Paradoxically, though, this does require analyzing details of courses and course sequences, and so to achieve this goal the approach taken is to evaluate the course sequences in the SE2004 model, by comparing them with a theoretical framework that identifies the stages by which students develop their ability to engineer software systems. This framework was developed by the author [6] from the work done with colleagues over the last 20 years to develop the undergraduate SE curriculum at the University of Sheffield.

A significant aspect of this work was ensuring that the curriculum made adequate provision for developing students' skills at software testing. Thus, the evaluation made here is based on comparing the ways in which software testing skills are developed in both the SE2004 and the Sheffield curricula, where the aim of the comparison is to validate the principles being discussed rather than the precise details of actual course content. This evaluation should therefore be seen as part of a much broader evaluation of the whole programme of action research based on the evolution of this curriculum, rather than as a report of the effects of any one particular step change to a part of this curriculum.

Hence, the structure of the rest of the paper is as follows. Parts II, III and IV describe respectively the key features of the SE2004 model, the theoretical framework and the Sheffield curriculum. Then, parts V to VIII discuss the four stages in the theoretical framework in turn, and finally part IX summarizes the conclusions of these discussions and provides an overall evaluation of this aspect of SE2004 and of the work done.

II. Knowledge and Course Structures

Both CS2001 and SE2004 use similar structures for defining their respective bodies of knowledge, in that they use a hierarchy of three levels, consisting of knowledge areas at the top level, knowledge units as the components of these and then topics as the components of the knowledge units. The same structure is also used in the revised version of CS2001 that was created in 2008 [7], and that is referred to as CS2008 from now on. Beyond this, though, there are some important differences of detail, in that both CS2001 and CS2008 define the levels of knowledge that should be achieved in terms of example learning outcomes for each knowledge unit, whereas SE2004 defines them in terms of a subset of Bloom's original taxonomy [8], which just uses the levels of knowledge, comprehension and application. As a consequence of this, CS2001 and CS2008 do not define explicit learning outcomes for the courses, since these follow from the knowledge units that are covered, and so neither does SE2004 for the courses that it imports from CS2001, but it does define such outcomes for the other courses that it describes. Also, whereas CS2001 simply numbers the units within each knowledge area, CS2008 gives each of them a code name that is derived from the title of the unit.

SE2004 then imports into its body of knowledge some or all of the core units from five of the CS2001 knowledge areas, although in each case it lumps all of the units from one knowledge area together into a single topic within its knowledge unit for CS foundations. This means that the underlying structure of these units and their topics is lost, unless one refers back to CS2001 for it, and it also means that a single Bloom level is assigned to all of the material of the units from one of these imported areas.

As far as software testing is concerned, the basic material on it that is imported in this way comes from two areas that only changed a little in the CS2008 revision: *Programming Fundamentals* and *Programming Languages*. This material is focused strongly on tying together the activities of testing and debugging, and so all of the relevant knowledge units have learning objectives of the form "design, implement, test and debug some kind of programming construction". Thus, for the Programming Fundamentals area, in the unit that CS2008 calls *PF/FundamentalConstructs* (PF1 in CS2001) these constructions are simple imperative programs; in the unit *PF/AlgorithmicProblemSolving* (previously PF2) they are more abstract algorithms; in *PF/Recursion* (previously PF4) they are simple recursive functions and procedures; and in *PF/EventDrivenProgramming* (which was PF5) they are simple event-driven programs. The one unit that did change in the revision was the one dealing with object-oriented programming, where some of the basic material was obviously meant to move from the *Programming Languages* area to *Programming Fundamentals*, to form a unit called *PF/ObjectOriented*. In this the relevant constructions are programs in any object-oriented language, with an emphasis on the role of inheritance relationships, but the same material and learning outcomes also still appear in the *Programming Languages* area as well, in the unit that was previously PL6 and is now coded *PL/ObjectOrientedProgramming*, which suggests that perhaps the revision process was not properly completed for this unit. Finally, there is also the elective unit in the *Programming Languages* area that was PL7 and is now *PL/FunctionalProgramming*, where the constructions are (obviously) programs that use the functional paradigm.

For the course structures, both CS2001 and CS2008 describe a number of alternative sets of introductory courses, to match the various possible strategies for introducing the programming paradigms: imperative first, objects first, functional first, breadth first, algorithms first or hardware first. Each of these sets of courses covers the various fundamental programming concepts, including testing and debugging, but their different starting points mean that they do so in different orders, and they do not define fully the ordering of all the later concepts. Hence, there is no one sequence defined explicitly through which the ability to carry out these various types of testing is developed, although the different sets all appear implicitly to progress in a roughly similar order, namely imperative constructions, algorithms, recursive constructions and then object-oriented ones. The variations then concern where event-driven programs and functional constructions are fitted into this, or (since the functional programming unit is an elective one) whether functional constructions are covered at all.

The course structures in SE2004 then define that any of these sets of introductory courses from CS2001 could in principle be imported into it, but in practice SE2004 also defines a set of introductory courses of its own, namely the sequence SE101, SE102 and SE200, which are intended to be used in place of these imported course sequences. This set too does not define precisely the order in which the different types of testing should be covered, and it does not make any mention of functional constructions, but it does imply a similar order to the one described above for most of the CS sets of courses.

Going on beyond foundational material, SE2004 then has a *Software Verification and Validation* knowledge area, which consists of two units that focus on software testing and three that cover the broader aspects of verification and validation. Of the two units that focus on software testing, the one that is simply coded *VAV.tst* consists almost entirely of core topics, and focuses primarily on functional testing, although it also includes topics for aspects such as testing across quality attributes and testing tools. The other unit, coded *VAV.hct*, also consists almost entirely of core topics, and it focuses on HCI testing and evaluation, and particularly on the aspects of usefulness and usability, as well as on testing techniques such as cognitive walkthroughs and methods for testing web sites. These two units are then complemented by the other three, of which one (coded *VAV.fnd*) covers terminology and foundations; another (coded *VAV.rev*) covers reviews; and the third (coded *VAV.par*) is concerned with approaches to analyzing and reporting problems at system level during the development process.

As well as this structure, CS2001 also has a *Software Engineering* knowledge area, and this has a unit that is coded SE6 and is concerned with software validation, which in CS2008 became *SE/SoftwareVerificationValidation*. The significance of this change of name is that the CS2001 version simply covered a few topics taken from the *VAV.tst* area in SE2004, namely: black box and white box testing techniques; testing at unit, integration and system levels; and object-oriented testing. Consequently, most of the sets of introductory courses in CS2001 include at least one hour of the core material from this unit, although since course content is only defined in terms of units rather than topics it is not clear exactly which of the topics is meant to be included in each course. In CS2008 the list of topics to be covered in this course was extended to also include different kinds of testing, such as HCI and usability testing, and reliability and security testing; and other roles for testing in the lifecycle, such as regression testing. The revision did not, however, involve updating the course descriptions, and so it is simply left to individual instructors to determine which of these additional topics might be included in any of the sets of introductory courses.

In terms of course sequences, SE2004 defines two alternative packages of core courses to follow on from the introductory sequences, and so virtually all of the topics from the *Software Verification and Validation* knowledge area have to be fitted into these. Each of these packages consists of six courses, which are structured into three stages, so that they can therefore be fitted into successive semesters or similar time periods.

The essential difference between the two packages is that package I takes a bottom-up approach to developing software systems, by starting with a course called *Software Construction* (SE211), while package II takes a top-down approach, starting with a course called *Design and Architecture of Large Software Systems* (SE213). Neither of these two initial core courses covers any topic from the unit *VAV.tst*, and package I leaves functional testing until the second of the three stages, in a course *Software Quality Assurance and Testing* (SE321) that puts this topic in the more general context of quality assurance. By contrast, package II treats this topic earlier, in a course simply called *Software Testing* (SE221) that is also in the first of the three stages, and which focuses more on the relationship between testing and requirements.

Each of these two courses (i.e. SE221 and SE321) does, however, only cover about two-thirds of the core topics. This is because there is another course, which is common to both packages and is called *SE Approach to HCI* (SE212), and this covers all of the core topics in the unit *VAV.hct*. Finally, the remaining core topics from *VAV.tst* are defined as being covered in the *Capstone Project* course (SE400). As its name suggests, this is intended to be the final course that students take before graduation, and the description of the course indicates that there should not normally be any formal lectures for it, so as to ensure that students are given adequate time to actually work on the project. This does, therefore, leave open the question as to whether placing topics here is really putting them rather too late in the students' experience to be of much practical value.

III. Development of Knowledge and Skills

The basic theoretical framework that is used here to describe the development of students' knowledge and skills within SE, and particularly the development of their ability to engineer software systems, was proposed originally in section 8 of [6]. The details of this framework were then developed further in [9], which used it to analyze the

treatment of software design skills in a similar fashion to the way in which the development of software testing skills is being analyzed here.

The framework is based on two features of the software systems that are being engineered, where one feature is the kinds of issues that the students should consider when developing them. This leads to the distinction between what the framework calls software development (SD from now on) and SE itself, where SD is defined as a very restricted subset of SE. The main feature of this subset is that SD only considers those issues that are relevant to the production of a system that will meet a given set of functional requirements from within a basic application domain (meaning, one that does not impose particular quality requirements on either the system or the process by which it will be developed). Hence, SD only considers the relevant structural aspects of SE, so that all of its quantitative aspects are either ignored or reduced to a minimal qualitative level. Then, once students have achieved some ability at SD, they can progress on to dealing with these aspects.

The other feature is the scale and size of the systems being produced, which identifies two stages of development within SD and two more within SE. The first of these, stage zero, is concerned simply with basic programming, as needed to produce a single program that performs a single well-defined function. Stage one (the actual SD stage) extends this to systems that are larger in scale, in that they may provide several functions, usually involving some kind of persistent storage. More importantly, SD also involves the complete software development lifecycle, rather than just the construction activity, so that the functions will need to be analyzed and modeled and the systems designed to provide them, as well as being implemented. Stage two then introduces the various quantitative issues that characterize SE, but without necessarily increasing the scale of the systems that students are able to engineer, which typically will be very small by comparison with typical industrial projects, as discussed in [10].

The final stage is therefore the professional development that is needed to scale up the skills learnt in an undergraduate course to the kinds and sizes of systems with which professional software engineers are concerned, as these are defined for instance in the Guide to the SE Body of Knowledge [11] (SWEBOK from now on). This phase is obviously outside the limits of an undergraduate curriculum, although of course it has an important influence on the design of such curricula, since an important requirement for them is that they must prepare students adequately for this stage. On the other hand, postgraduate courses in SE do fit within this fourth phase, and for them the Graduate SE 2009 Curriculum Guidelines [12] (GSwE2009 from now on) play a similar role to that played by SE2004 for undergraduate curricula. Thus, while the primary focus of this paper is on evaluating the SE2004 model, it is also appropriate to discuss the relationship between this theoretical framework and the GSwE2009 model, which is done in part VIII.

As this framework was originally described, it was significant because students could be expected to develop their skills in a simple progression through the four stages. While this is still the case, more recent development of the framework has identified two other significant relationships between its stages. One of these is the relationship between the SD and SE stages, where [13] demonstrated, from an analysis of earlier curriculum models, that SD was more than a stage on the way to developing students' skills at SE. This was because, for any of the branches of computing that are identified in the overview report for the Computing Curricula 2001 project [14] (CC2005 from now on), SD as described in this framework characterizes the set of skills relating to the development of software systems that students in those branches of computing need to master in order to operate effectively.

The other significant relationship between stages in this framework is the one between the programming and SD stages. While these two stages represent a progression, in the sense of widening the scope of the artifacts that are being developed, from programs to systems, [15] showed that they are also complementary. This is because these two stages can be seen as representing the pure and applied aspects of computing respectively, and so are in a relationship that is similar in its nature to that between pure mathematics and applied mathematics, with the result that there are some dependencies in both directions between them.

As well as this framework, that is at least generic to the whole of SE, and where some parts of it apply to the whole of computing, for the specific topic of software testing there is also a second theoretical framework that has emerged within the research community. This second framework identifies different levels of maturity for the activity of testing, based on the purpose for which it is performed, although these levels also reflect the historical development of testing. As characterized by Ammann & Offutt [16] there are four of these levels, where level 0 treats testing as part of debugging; level 1 identifies its purpose as demonstrating the correct operation of systems; in level 2 this becomes finding faults in systems; and level 3 views it as reducing the risks associated with systems, by identifying both situations where they operate correctly and ones where they exhibit faults. Thus, each of these levels of maturity depends on the ones below it, and in particular they all naturally relate to debugging, since at any level of maturity, once testing has found a fault then the obvious thing to do is to try to remove it, which is the purpose of debugging.

IV. The Structure of the Sheffield Curriculum

This theoretical framework has been both derived from and used to drive the development of the Sheffield curriculum, and in particular its key structural feature, which is that the curriculum is based on a "spine" that consists of a major project in each year, with the various courses being fitted round this spine so that they provide the knowledge that students will need in order to undertake the projects [17]. The project that is used in the first year has been described elsewhere [18], and takes students through the complete development lifecycle in a way that is very closely integrated into the initial courses on SE, and that uses academic staff to play the roles of clients for the systems being created. In the second year the project, which involves

teams of students competing to build real systems for external clients, is much more loosely structured, in that it forms a course on its own. This takes up one third of the second semester, and it follows on from courses covering essential technologies, such as databases and HCI.

For the third year the project is an individual capstone-style one, since in the UK the third year is the final one for students who will graduate with a bachelor's degree, and this too forms a course on its own. The fourth year structure (for those who continue to it) is at masters' level, and the project component of it essentially involves the students in running a software house, known as Genesys Solutions [19]. Hence, they not only have to undertake and manage the team projects that are carried out by the software house, but also manage its portfolio of projects. This management activity therefore involves both the business aspects of balancing resources across the teams, and the technical aspect of identifying and managing opportunities for reuse of components and other concepts across projects.

The effect of this key structural feature is that the curriculum as a whole uses the approach of problem-based learning, even though individual courses within it may not all use this approach. In particular, this curriculum structure emphasizes the principle of setting the knowledge that students must acquire within its practical context, so that they are better able to understand it by seeing its application to the problems on which they work. The realism of these problems then helps the students to develop the skills needed to apply this knowledge effectively.

The relationship between this curriculum structure and the overall theoretical framework is that stages zero (programming) and one (SD) are developed in parallel pairs of courses in the first year, with some advanced aspects of the programming continuing in second year courses. This parallel development is possible because much of SD consists of activities other than programming, so that the two only need to link together in the construction stage of the first year project, part way through the second semester. Because there are no external clients involved, the scope of this first year project can be restricted to just meeting basic sets of functional requirements, so that it falls entirely within SD.

By contrast, the second year courses start to introduce some of the qualitative aspects of SE, such as the efficiency of databases, the quality of designs and the usability of interfaces, and so for the theoretical framework these courses mark the transition from stage one (SD) to stage two (basic SE). These qualitative aspects are needed to prepare the students for the second year projects, because these involve teams competing to produce systems that best meet the clients' requirements, which are not just functional, so that qualitative assessments are essential for comparing what each team produces. Hence, these projects do require the application of real SE principles and practices, even though students' knowledge of these is by no means complete. This usually becomes very apparent to them during the projects, and provides the motivation both for the specialized third year SE courses and for the further development of their skills during the third year individual projects.

For those students who continue into the fourth year to study at masters' level, the transition from the guidelines in the SE2004 model for undergraduate programmes to those in GSwE2009 for postgraduate programmes also represents the transition from stage two of the theoretical framework (basic SE) to stage three (professional development). As such, the significance of this stage to the evaluation of the SE2004 model, which is the purpose of this paper, is that it may provide evidence of aspects where the previous study might not be preparing students adequately for the work that they will do at this stage, and this is the focus of the discussion in part VIII of the paper.

This therefore completes the description of the background material for this paper, namely the SE2004 model, the theoretical framework against which it is to be evaluated, and the Sheffield curriculum from which this framework has been derived. The rest of the paper is then concerned with analyzing how well the coverage of software testing in the SE2004 model fits in with each stage of the framework. These stages are considered in turn, beginning in the next part with the programming stage.

V. Programming and Software Testing

At the programming stage, the assumption in the theoretical framework is that testing is concerned essentially with single test cases, and that the most important aspect of an arbitrary test case is the way in which the choice of inputs for it will cause the program construction being tested to follow a particular execution path. At this stage, therefore, the key measure of whether or not a test case is successful is whether or not this execution path is the intended one, and the output that is produced from the execution is seen primarily as the visible indication of this test result. Consequently, in terms of the separate framework for the maturity of software testing activities, this treatment is located firmly at level 0.

In the SE2004 model, the practical effect of the linkage between testing and debugging in the material imported from CS2001 is that the same approach implicitly applies to all the various forms of testing that it covers, and hence to each of the kinds of constructions that are covered in the SE2004 introductory courses. This includes those object-oriented structures that involve a number of objects of different classes, so that they are beginning to approach the complexity of the systems with which SD is concerned. Hence, such material on testing as is included in the courses *SE and Computing III* (SE200) or *Introduction to SE* (SE201), which form the ends of the specialized and imported introductory sequences respectively, still implicitly takes the same approach as in the programming stage of the theoretical framework. Consequently, these introductory sequences do both cover effectively all of the software testing material that would come in this stage.

On the other hand, there is an issue as to where in the course sequences the boundary should come between the programming and SD stages. Currently the situation is simple: the only testing material in the SE2004 introductory course sequences is that which the theoretical framework would describe as belonging to the programming stage. By

contrast, SE2004 specifies that each set of introductory courses should cover enough of other key topics (such as requirements analysis, system design and basic development processes) as to indicate clearly that the material covered in these courses should fit into the SD stage in the theoretical framework.

Hence, for the specific topic of software testing there appears to be a mismatch between the SE2004 model and the theoretical framework, in that the model only covers material from the programming stage in its introductory courses. This means that if there is additional material on software testing that would relate to the SD stage of the framework, then this is not covered in these courses, even though they do cover SD material for other topics. The next part therefore discusses what material on software testing belongs to the SD stage, and whether any of this ought to be in the introductory courses rather than in the core courses that follow on from them.

VI. The Testing Component of Software Development

Since the SD stage of the framework is concerned with creating systems that just meet sets of basic functional requirements, the underlying purpose of software testing at this stage has to move up at least one level of maturity in the separate framework for software testing. Thus, instead of being tied to debugging, testing now has to be concerned at least with demonstrating the correct operation of the systems, and possibly also with finding faults in the early versions of them where these do not operate correctly.

This then has two implications for how software testing should be treated at this stage of the framework. One is that the focus on single test cases has to be broadened, to cover the creation, use and management of sets of test cases. The other is that the focus on the kind of testing being performed has to be broadened from just unit testing, to also cover integration testing and system testing. Here, though, the scope of system testing will of course be restricted to just testing for functional correctness rather than for any aspects of system quality, although the need for a system to meet at least some minimal requirement for usability, so that it can actually be seen that the functional requirements are being met, means that some attention will also need to be given to very basic usability testing.

This broadening means that in principle there are three sets of topics from the *VAV.tst* unit of the SE2004 model that are needed for this stage, and it also means that students need to be able to apply the techniques covered in these various sets of topics. One set consists of the topics that cover the concepts of test sets and the methods for creating them, namely *VAV.tst.3* for structural testing and *VAV.tst.4* for functional testing. The second set contains those concerned with testing larger scale constructions than code units, viz. *VAV.tst.5* for integration testing (although the model only requires students to understand this material, rather than be able to apply it), together with some material from *VAV.tst.6* for testing based on use cases (or their equivalents) and from *VAV.tst.8* for system and acceptance testing. The third set of topics is concerned with the management of test sets, and so it consists of basic material from *VAV.tst.10* for regression testing (although again the model only expects students to understand this material, rather than be able to apply it), and from *VAV.tst.11* for testing tools.

Also, the need for some consideration to be given to basic usability testing means that at this stage in the framework there has to be some coverage of material from the *VAV.hct* unit of the SE2004 model, so as to cover basic human-computer interface testing. Specifically, two topics are needed, where the first (*VAV.hct.1*) covers the concepts of usefulness and usability, and the model simply requires that students know about these concepts. Then, the topic *VAV.hct.4* covers approaches to carrying out user testing, and both the framework and the SE2004 model need students to be able to apply a suitable approach for doing this.

The SE2004 model does not define these various sets of topics in much more detail than has been done here, since this detail is essentially documented in the SWEBOK. What emerges from the descriptions there is that there are actually progressions within each of these sets of topics, and educators need to be aware of these, even though they are not specified by either the SE2004 model or by either of the two theoretical frameworks.

Specifically, in the first set, which is concerned with creating test sets, there is a progression within structural testing: from the basic concept of structural units that can be covered by test cases, to methods for constructing test sets iteratively by identifying at each iteration those units that have not yet been exercised. Similarly, there is a progression within functional testing: from the basic concepts of equivalence partitions, their boundaries and the constraints between them, to methods for analyzing sets of equivalence partitions (ie test frames), and then generating test sets from these.

Furthermore, for each of these forms of testing there is also a progression from learning about one such method to comparing several alternative methods for constructing test sets. At the SD stage the restriction to ignoring issues of quality means that it is sufficient to ignore this progression, so that at this stage of the framework it would be sufficient to just cover one structural method of testing and one functional method. The structural method would almost certainly involve focusing on one of the basic types of structural elements (such as branches) and trying to ensure adequate coverage of these, while for a functional method the best established choices are either the category-partition method or the classification tree method [16 p166]. Here, the reason for covering two types of method is not to compare them, but because of their relationships with other topics, since the structural methods will develop naturally from the previous links between testing and debugging, while the functional methods will link with the topics of system and acceptance testing.

Within the second set of these topics, concerned with testing larger constructions, the progression is implicit in the increasing scale of them, from units through sub-systems to complete systems. Again, though, the problems associated with these different kinds of constructions link to other topics, and so each of these elements in the progression

needs to be covered at the SD stage. Within the third set of topics, concerned with the management of test sets, the progression is from running tests once to running them multiple times as changes are made to material that has been tested. The significance of this is that it justifies at least introducing tools such as the XUnit ones [20] to automate this process, and so again provides a natural linkage between these two topics.

For the SE2004 courses, the coverage of these topics all occurs in either of the two alternative courses *Software Testing* (SE221) or *Software Quality Assurance and Testing* (SE231), as described above in part II. There is, however, no indication that these topics are even meant to be touched on in any of the sequences of introductory courses – not even in the last courses of the sequences, viz. *SE and Computing III* (SE200), or the equivalent alternative *Introduction to SE* (SE201) that follows on from any of the imported pairs of CS101 and CS102. Hence, for software testing the SD stage of the theoretical framework is not reached at all in the introductory courses, so that students only progress to it once they are some way into the core courses.

This situation contrasts sharply with the Sheffield Curriculum, where it has been found essential to cover at least the basics of all of these topics within the pair of first year courses that introduce SD. The reason for needing to cover this material there is that the project incorporated into these courses involves teams of students in creating complete systems, and so the final stage of this project has to focus on them determining whether the systems that have been created do actually meet their requirements. Hence, the students need to be able to test these systems in a reasonably systematic fashion: certainly far more systematically than they would be able to do if they had not been taught anything about structural, functional, interface and system testing, or about the use of appropriate testing tools.

The need to accommodate this (and other) material has meant that the introductory sequence in this curriculum actually consists of four courses rather than the three described in SE2004. The benefit of this is, however, that by the end of this longer introductory sequence students have covered the SD stage of the theoretical framework, whereas in the SE2004 model they will not even have started on the software testing material needed for this stage.

This therefore means that, from the perspective of this theoretical framework, there is some inconsistency in the SE2004 model, arising from the different treatments of software testing as compared with other key activities in the software lifecycle. In particular, for software design the analysis in [9] showed that a significant amount of material relating to it that is covered in the introductory courses does fit into the SD stage of the framework, although it would be better if more material needed for this stage could also be covered in those courses. By contrast, for software testing these courses do not cover any material that belongs to the SD stage.

Hence, the effect is that, while some topics in software design that belong to this stage are covered in the introductory courses, topics for software testing (which is an equally key activity in SD) are deferred until the core courses. This represents a definite mismatch between the SE2004 model and the theoretical framework, in that while they both recognize an important boundary between stages, which for the SE2004 model comes between the introductory and core courses, and for the theoretical framework comes between the programming and SD stages, they do not put this boundary in the same place.

VII. The Testing Component of Software Engineering

In the theoretical framework, the key feature of the progression from SD to SE is that issues relating to quality are introduced, including approaches to measuring and managing quality. This broadening of the approach therefore needs to apply just as much to test sets as to actual software systems. Specifically, this means that whereas the SD stage introduces the concepts and methods (or techniques, as the SWEBOK calls them) that are needed for constructing test sets, it does not go as far as evaluating either the test sets that are produced or the methods by which they are produced. Of course, if SD is being taught through the use of realistic projects then the better students may well be motivated to start exploring for themselves how they might improve their test sets, and what the effects of doing so might be on the quality of the software that they produce. This motivation to explore SE concepts is, however, a benefit of this particular approach to teaching, rather than a principle for structuring what needs to be taught.

By contrast with this restriction on SD, at the SE stage the evaluation of both testing methods and test sets becomes of fundamental importance, since the focus is no longer just on how some test set can be constructed, but on how systematic the construction process will be, and consequently what properties the resultant test sets will possess. This change of focus then has three consequences for how the material relating to software testing should be organized at this SE stage.

The first consequence is that, whereas at the SD stage it is sufficient to introduce single examples of each of structural and functional test methods, at the SE stage it is important to expose students to a variety of methods for each kind of testing. Thus, for structural testing they need to move on from simple definitions of structural units that are derived from control flow, to the problems that arise when paths are to be considered, and hence to the more advanced kinds of structural units that are intended to solve such problems, and in particular those that are derived from concepts of data flow. Similarly, for functional testing students need to cover a variety of approaches to representing the combinations of equivalence partitions and the constraints between them, such as those based on logical or algebraic structures and those based on graphical or hierarchical structures.

The second consequence of needing to compare different test methods is that issues such as test selection criteria and test adequacy obviously also need to be covered at this stage in the framework, but they only need to be introduced here, rather than in the earlier stages. Hence, the requirements of the theoretical framework are quite different from those implied by the description of software testing in the

SWEBOK, where these issues appear as the very first key ones. This ordering of the material in the SWEBOK may be appropriate theoretically, in terms of the way in which it underlies the comparison of different test methods, but it is certainly not appropriate pedagogically, where being able to construct simple test sets is a far more basic learning objective than being able to compare them.

The third consequence of the change of focus at this SE stage in the theoretical framework is that there must be a progression in the approach taken to the purpose of testing, corresponding to moving further up the levels defined in the framework for the maturity of software testing. This means that from just being concerned with testing for functional correctness, as in the SD stage, at the SE stage it is also necessary to consider many of the other purposes for which testing may be conducted. The SWEBOK calls these objectives for testing, and they include performance testing, stress testing, usability testing, etc., and in the SE2004 model they form the topic *VAV.tst.9*. In the framework for the maturity of software testing these objectives correspond to the different kinds of risk associated with the operation of a system, since inadequate performance or usability are just as much risks as actual failures in operation, and so testing of a system has to be concerned with these issues. This therefore puts the activity at level 3 of this framework, whereas the testing for functional correctness that is required for SD is only at levels 1 or 2 in it.

This broadening of the scope of software testing at the SE stage also has the effect that it becomes increasingly linked with other knowledge areas, such as the Software Quality areas in both the SWEBOK and SE2004. For instance, functional testing and structural testing are known to be able to find different kinds of faults, since functional testing can identify unimplemented requirements whereas structural testing can not. Similarly, structural testing can identify code that is unrelated to any specified requirement, and so is implementing features that are at least unspecified and possibly also unwanted, since they might actually be potential security breaches, whereas functional testing can not identify such code. Consequently, one starting point for evaluating functional test sets is by using structural testing concepts to measure their coverage, and the author has found this a useful approach to take in linking software testing with the assessment of software quality. Similarly, other kinds of testing also involve the measurement of different quality attributes in each test, so that software measurement is also an important topic that needs to be introduced, which in terms of the SE2004 model forms the topic *VAV.fnd.4*.

This link with software measurement is another example of a topic being labeled as foundational within the SWEBOK because of its theoretical importance, when pedagogically it should be deferred until later in the curriculum. In this case, though, the pedagogical requirements are reflected in the SE2004 course structures, since the one core hour of lecture material that is allocated to this topic is covered in the course *SE Approach to HCI* (SE212). Of course, this does raise the issue of whether one core hour is enough time in which to provide proper coverage of software measurement, and the author's view would be that it is not, but this must be regarded as a separate issue, and one that can not be pursued further here.

VIII. Software Testing and Professional Development

The last stage in the theoretical framework for the development of skills and knowledge in SE goes beyond the basic ability to engineer simple systems, and involves developing this to match the requirements of SE as practiced professionally. It therefore involves learning to cope with two distinct aspects of the systems being engineered, one being their greater size and scope, and the other being the much wider range of issues that might need to be considered during their development. As has been noted in [10], there is a limit to the extent to which university degrees can provide experience with large and complex systems, and particularly in undergraduate curricula, which is why this level of the theoretical framework goes beyond the SE2004 model. On the other hand, university programmes at postgraduate level certainly can support students in broadening their skills to cope with the wider range of issues that arise when real systems are being developed, and indeed part of the rationale for the development of the GSwE2009 curriculum model was to show how this broadening of skills might be achieved.

This does not mean, however, that the discussion of this stage in the theoretical framework is irrelevant to the goal of this paper, namely the evaluation of the treatment of software testing in the SE2004 model, for an important issue is whether that model does provide students with an adequate basis for further developing their skills in this stage of the theoretical framework. Here, there are four main issues that need to be considered.

The most important of these issues is that GSwE2009 is not actually aimed at describing degree programmes that follow on directly from those described by SE2004. Rather, the starting point for GSwE2009 (as described in its chapter 4) is that students should have an undergraduate degree in computing, plus at least two years practical experience of doing some form of SE, so that in terms of formal education this is closer to CS2001 than SE2004. The model then defines (in table 1 of section 6.2) the levels of knowledge and skills that this starting point is intended to provide, but since for SE the only one of these that is even up to the application level in Bloom's taxonomy is for the knowledge unit *software construction*, it is generally requiring much less in the way of skills than the levels that SE2004 would expect graduates to reach.

This is particularly true for software testing, where the only topics that GSwE2009 expects students to have covered previously are fundamentals, test levels and test techniques, and then only to the knowledge level of Bloom's taxonomy. Indeed, section 6.1 explicitly states "*Students admitted to a program who possess substantial SwE education (e.g., a Bachelor of Science degree in SwE) … will arrive with knowledge at or above some of the designated Bloom's levels.*". In terms of the theoretical framework, this means that GSwE2009 actually defines its starting point as somewhere beyond the boundary between the SD and SE

stages, but closer to it than to the boundary between the SE and professional development stages, if one takes the latter as being defined by the end point of SE2004. Hence, it is very clear that the pre-requisite knowledge or skills for SE that are required by GSwE2009 are simply a sub-set of those provided by SE2004.

The second issue is then how the levels of SE knowledge and skills that form the end point of SE2004 relate to those that form the end point of GSwE2009. For each of these models the ranges of topics that are covered are derived from the SWEBOK, although in both cases topics have been added to those specified in it, where these extra topics are concerned either with foundational material or with non-technical issues. The foundational material is outside the scope of the SWEBOK but is needed for any curriculum model, and section 6.1 of GSwE2009 makes clear that the selection of it in SE2004 was a significant source. The non-technical issues focus on ethics and professional conduct, and again section 6.1 indicates that the choice of them was influenced by SE2004, and it also suggests that knowledge areas covering these issues are likely to be incorporated into the next version of the SWEBOK.

The third issue in comparing GSwE2009 and SE2004 is that, while in principle they both appear to be using the same definitions of the levels in Bloom's taxonomy when defining the levels of knowledge and skills that should be achieved, in practice there are some differences in the ways in which they apply these levels. Specifically, SE2004 only uses the levels of knowledge, comprehension and application, whereas one might expect that SE activities such as requirements analysis or software design would involve at least the analysis and synthesis levels respectively. This use of the levels was discussed at one of the workshops during the development of SE2004 [21], where the explanation given for not using the higher levels was that they were intended to refer to analysis or synthesis of knowledge, rather than of SE artifacts, and so it was not considered appropriate to use them. By contrast, the team developing GSwE2009 came to the opposite view, and their descriptions of how these levels could apply to SE artifacts are given in its appendix B. The consequence of this difference in interpreting the levels is that SE2004 appears to be requiring lower levels than would be implied by the GSwE2009 descriptions, but the practical effect of this is simply that graduates from programmes that conform to SE2004 will actually have exceeded the levels specified in SE2004. Hence, such graduates will also have exceeded the pre-requisite levels specified by GSwE2009 by more than the comparison of the two models would suggest.

The fourth issue in comparing these two models is that GSwE2009 puts a lot more emphasis on the significance of application domains than does SE2004, where consideration of them is essentially confined to the knowledge area *Systems and Application Specialties*. This difference results from the increasing recognition that, as discussed in [22], the choice of application domain does indeed have a wider impact than just the choice of certain specialties, as it may well affect such basic issues as: which models need to be created during requirements analysis; which of the various standard architectures or patterns must be considered during software design; or which non-functional properties should have the highest priority for testing. Since these kinds of impacts are now more widely recognized, it is likely that they will need to be given much more prominence when a successor to SE2004 is being created.

The overall conclusion to be drawn from these issues is therefore that the relationships between the SE2004 and GSwE2009 models are actually not as close as one might expect for undergraduate and postgraduate curricula in the same discipline. This is mainly because GSwE2009 does not take the end point of SE2004 as its starting point, but only requires much lower levels of SE knowledge and skills, as represented by models such as CS2001 and the volumes for the other disciplines within computing that are described by CC2005. Apart from this difference, however, there are no other significant mismatches between the two models.

IX. Conclusions

The main conclusion of this paper is that the treatment of software testing within the SE2004 course structures does not provide as well as it should for the progressive development of students' ability to test software systems. Specifically, while the model specifies appropriate material in the relevant knowledge units, the division of this material between the introductory and core courses means that the former only cover those aspects of software testing that are related to programming. All of the material that relates to both the SD and SE stages of the theoretical framework is then allocated to core courses, in a way that does not recognize at all the progression from SD to SE.

Providing better progression would therefore need significant modification to these course structures, as the natural correspondence between SE2004 and the framework would imply that the introductory courses should cover the SD stage of the framework and the core courses its SE stage. To achieve this correspondence for software testing would involve moving a significant amount of material on testing from the core courses to the introductory ones, which is easy to propose as a theoretical conclusion, but if the proposal is to be useful it requires some discussion of how it could be achieved in practice.

The key point here is that curriculum design is fundamentally constrained by the number of hours available, so that normally any proposal for adding new material has to be accompanied by a proposal for what should be left out to make room for it. In this case, however, what is being proposed is not the addition of extra material – the topics on software testing needed for the SD stage of the theoretical framework are already in the SE2004 model – but rather the movement of this material from one place in the curriculum to another. Thus, the overall effect of the changes being proposed here for material on software testing, together with those proposed in [9] for material on software design, would lead to the introductory sequences having to be extended by one course, but would also lead to one fewer core courses being required.

Of course, any claim that such a change would be an improvement must be validated, but validating a comparison of complete curriculum structures is extraordinarily difficult,

as one university would not have the resources to run two curricula side by side, while any attempt to compare different universities would have too many uncontrolled factors to be useful. In this case, though the validation comes from the fact that the solution has already been shown to work, since the Sheffield curriculum has evolved to incorporate an extra introductory course along precisely the lines being proposed here, and for the reasons that have been discussed above. This curriculum has been developed through the process of action research discussed in the introduction to this paper, and so it has been validated by the series of incremental steps resulting from this process, which therefore validates the conclusion that such a development from the SE2004 course sequences would be a valuable one.

Against this, it could perhaps be argued that the Sheffield curriculum has other particular features, such as the focus on project-based courses, which means that it is might be these other features that have been validated rather than this particular change to the introductory course structure. Part of the reason for developing the theoretical framework used in this paper was to enable such arguments to be countered, since the framework was derived through the same process of action research, in order to give a way of abstracting out the principles that were being identified from the details of any one particular curriculum.

The key principle being identified here is that the distinction between SD and SE provides a natural basis for describing the level of knowledge and skills (particularly skills) at SE that students should have achieved by the end of the introductory sequence of courses. This then determines what SE topics students should cover in the introductory courses, and the contribution of this paper has been to analyze this for the specific activity of software testing, and to do so in a way that would enable any SE educator to identify how they might achieve similar improvements in their introductory course sequences. Making such a change would not be dependent on the adoption of project-based courses – although we would certainly argue that such courses help to improve the balance of knowledge and skills – but the change would make much clearer what levels of SE skills students should be aiming at. This additional clarity of learning outcomes, over and above those already defined in the SE2004 model, is the main benefit that would result from applying the ideas presented in this paper.

REFERENCES

[1] IEEE-CS & ACM Joint Task Force on Computing Curricula, "Computing Curriculum - Software Engineering: Final Report" (21st May 2004).

[2] IEEE-CS and ACM Joint Task Force on Computing Curricula, "Approved Final Draft Version of Computing Curricula 2001" (15th December 2001).

[3] Computing Curriculum for SE Steering Committee, "Review Comments for CCSE Volume (first draft - 7/17/03)", <http://sites.computer.org/ccse/FirstReviewComments7-17-03.pdf>.

[4] Computing Curriculum for SE Steering Committee, "Review Comments for SE2004 Volume – Public Draft 3 - 1/25/04)", <http://sites.computer.org/ccse/Review2&3Comments5-1-04.pdf>.

[5] M. Sahami, M. Guzdial, A. McGettrick & S. Roach, "Setting the Stage for Computing Curricula 2013: Computer Science – Report from the ACM/IEEE-CS Joint Task Force", Proc. SIGCSE Technical Meeting 2011, Dallas, Texas, 161–162 (2011). Doi: 10.1145/1953163.1953213.

[6] A. J. Cowling, "Modelling: A Neglected Feature in the Software Engineering Curriculum", Proc.16th Conference on Software Engineering Education and Training, IEEE Computer Society Press, 206–215 (2003). Doi: 10.1109/CSEE.2003.1191378.

[7] ACM & IEEE-CS Interim Review Task Force, "Computer Science Curriculum 2008: An Interim Revision of CS2001", (December 2008), <http://www.acm.org/education/curricula/ComputerScience2008.pdf>.

[8] B. S. Bloom (ed), "Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook 1: Cognitive Domain", McKay, 1956.

[9] A. Cowling, "Stages in Teaching Software Design", Proc.20th Conference on Software Engineering Education and Training, IEEE Computer Society Press, 141–148 (2007). Doi: 10.1109/CSEE.2007.47.

[10] A. Cowling, "What Should Graduating Software Engineers Be Able To Do?", Proc.16th Conference on Software Engineering Education and Training, IEEE Computer Society Press, 88–98 (2003). Doi: 10.1109/CSEE.2003.1191354.

[11] P. Bourque & R. Dupuis (editors), "Guide to the Software Engineering Body of Knowledge, 2004 Version", IEEE Computer Society Professional Practices Committee (2005).

[12] Integrated Software and Systems Engineering Project, "Graduate Software Engineering 2009: Curriculum Guidelines for Graduate Degree Programs in Software Engineering", Stevens Institute of Technology (2009), <http://www.gswe2009.org>.

[13] T. Cowling, "Software Development as the Core of Informatics", Proc. 3rd Informatics Education Europe Conference, Università Ca' Foscari, Venice, 155–169, (2008), <http://www.dsi.unive.it/IEEIII/atti/PROCEEDINGS_IEEIII08.pdf>.

[14] The Joint Task Force for Computing Curricula 2005, "Computing Curricula 2005: The Overview Report", ACM Press (2005), <http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf>.

[15] A. J. Cowling, "Pure Versus Applied Informatics", Proc. Eqanie Conf. Learning Outcomes and Quality Management in Informatics Education, Vienna, Austria, <http://www.eqanie.eu/pages/events/conference-vienna-2011/proceedings.php>.

[16] P. Ammann & J. Offutt, "Introduction to Software Testing", Cambridge University Press, 2008.

[17] A. J. Cowling, "The first decade of an undergraduate degree programme in software engineering", Annals of Software Engineering 6, 61–90 (1998). Doi: 10.1023/A:1018940911475.

[18] A. J. Cowling, "The Crossover Project as an Introduction to Software Engineering", Proc.17th Conference on Software Engineering Education and Training, IEEE Computer Society Press, 12–17 (2004).

[19] Genesys Solutions, <http://www.genesys-solutions.co.uk>.

[20] Object Mentor, "Resources for Test Driven Development", <http://www.junit.org/>.

[21] S. Mengel et al, "IEEE-CS/ACM Computing Curriculum Software Engineering Volume Project", Proc.16th Conference on Software Engineering Education and Training, IEEE Computer Society Press, 333–334 (2003). Doi: 10.1109/CSEE.2003.1191398.

[22] A. J. Cowling, "The Role of Application Domains in Informatics Curricula", Proc. 2nd Informatics Education Europe Conference, Thessaloniki, Greece, SEERC, 166–175 (2007), <http://www.seerc.org/ieeii2007/>.