Int. J. Engng Ed. Vol. 10, No. 5, pp. 416–426, 1994
Printed in Great Britain.

0742–0269/91 $3.00+0.00
© 1994 TEMPUS Publications.

Experiences Gained in the Teaching of Computing Technology to Engineering Students*

R. L. WOOD
L. E. DAVIS
S. T. NEWMAN
K. R. HOLMES
Department of Manufacturing Engineering, Loughborough University of Technology, Loughborough, Leicestershire, LE11 3TU, UK

This paper discusses the experience gained by staff and students through the recent replacement of a traditional computer programming course for manufacturing engineering students by a more broadly based 'Computing Technology' course. These courses differ in that the latter additionally includes software engineering and an introduction to various manufacturing applications of computing technology. By placing the course within the contexts of industrial need and of the degree courses to which it contributes, the motivation and objectives of the course are compared against the outcome of its introduction in the 1991/92 academic year. It is concluded that the course is a significant improvement on the more traditional and highly focused approach to teaching computer programming. Not only does it facilitate the writing of more structured and better quality software, but it has also created many opportunities for developing a deeper understanding of computer-based technologies.

INTRODUCTION

THE DEPARTMENT of Manufacturing Engineering at Loughborough University of Technology offers three undergraduate and one postgraduate degree courses that integrate manufacturing engineering, design and management topics. The aim of these courses is to produce young manufacturing and design engineers who have significant potential for future technical management positions. Education in the use of a computer programming language has been an integral element of these courses for many years. Recently, however, this programming course has been replaced by one, entitled 'Computing Technology', that additionally covers software engineering techniques and other material to provide a wider awareness of the industrial development and use of computer-based systems.

Three factors have motivated recent developments [1]:

1. Traditional 'computing' courses have essentially remained static for many years. However, within industry an increasingly sophisticated approach has evolved to the development of engineering software, based on various systems analysis and design techniques such as Yourdon [2], IDEF [3] and SSADM [4].
2. An understanding of the systems approach to managing complexity is of much wider application than in the creation of computer-based systems. Thus, engineering students should benefit by obtaining such understanding as soon as possible.
3. Recognition that many computer-based systems introduced into manufacturing industry fail to provide all of the benefits envisaged during their conception. Typically, the reasons for this are twofold. Firstly, those manufacturing engineers involved with the initial specification of such systems have little awareness of the complex interactions that occur between large-scale computing technology and manufacturing systems. Secondly, there is a general lack of common understanding between the vendors and the intended users of such technology.

On the undergraduate programme, the Computing Technology course is presented throughout the first year of study and constitutes one out of 12 modules. In the introductory year, 1991/92, 70 students took part. On the postgraduate programme, the course represents one-half of a module (of which there are 12), taken over the Autumn term only. In the same introductory year 30 students took part in the M.Sc. variant.

This paper briefly reviews the objectives and structure of the course. Initial results and an evaluation of the course are presented, followed by discussion of recent and possible future developments, and spin-off activities.

* Accepted 17 July 1994.

OBJECTIVES

The Computing Technology course was designed to fulfil the following objectives:

- To educate students in structured techniques for the analysis, design and specification of systems, exemplified by the analysis and design of computer-based systems.
- To introduce students to a high-level programming language from an analytical standpoint, changing the emphasis in programming from the creation of a solution to the translation of an engineering solution to some problem. The primary focus in this is to teach language facilities on a 'need-to-know' basis.
- To provide future manufacturing and design engineers with a broad introduction to computing technology and its enabling role within industry.
- To develop an approach to achieving the above objectives that is common to teaching at both undergraduate and postgraduate levels.

COURSE DESCRIPTION

In order to meet the above objectives, a course structure was developed that included software engineering techniques, a computer programming language and various computer applications that had a manufacturing focus.

Syllabus

In the introductory year the contents of the lecture course were as follows:

- Software engineering
- Programming language
  postgraduate: ANSI C; undergraduate: Pascal
- Applications lectures introducing:
  computer hardware and software
  computer-aided engineering
  computer-aided manufacture
  applications development software
  office automation
  simulation
  real-time systems
  vision systems

Figure 1 shows the undergraduate course timetable in the introductory year.

The adopted software engineering methodology was based on the Yourdon technique. It was considered that the insight provided by this technique in making a clear distinction between the structure and dynamics of systems, along with structured relationships between data items, would maximize its utility across the widest possible range of topics within the degree syllabuses.

The programming languages were selected on the basis that they should provide a logical transition from engineering solution to implemented program through block structure for code and flexible structuring and abstraction facilities for data.

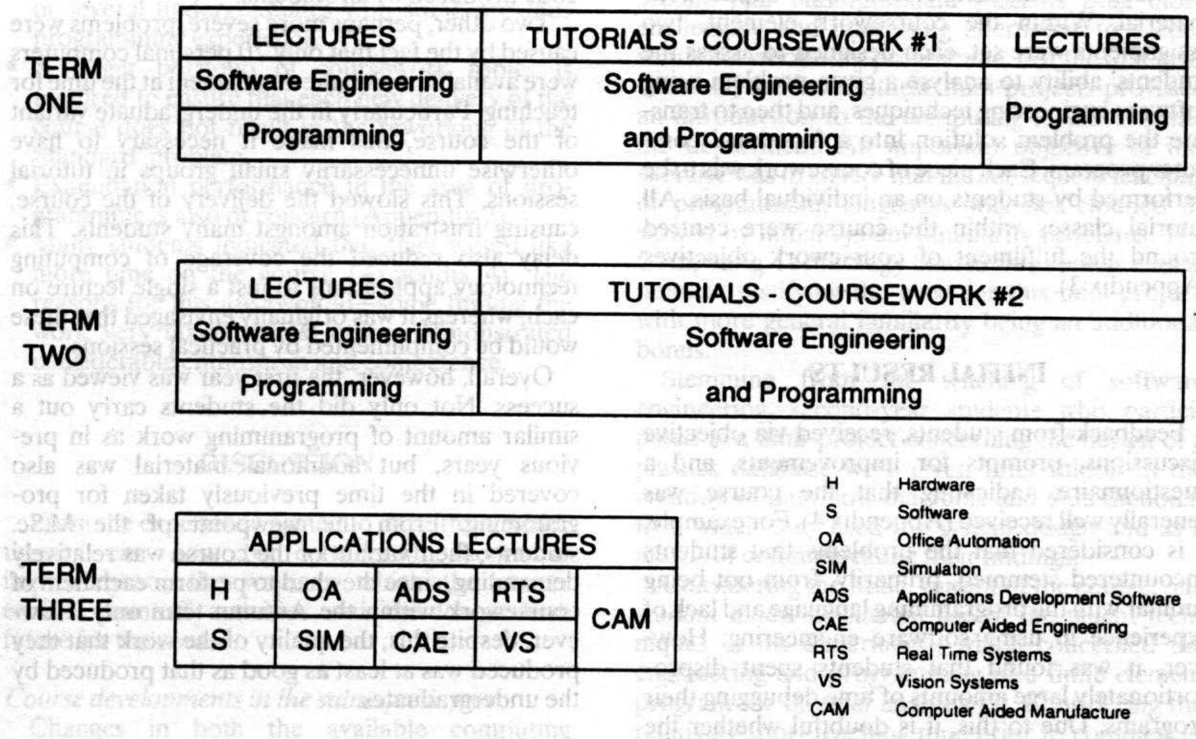

Fig. 1. Computing Technology timetable (undergraduate).

The C language was selected for the M.Sc. course for reasons in addition to those above. Firstly, there are many opportunities for M.Sc. students to carry out their dissertations in conjunction with ongoing research within the department. A significant feature of much of this research is system and software development, mostly using C. Secondly, C is becoming more widely adopted within industry. Thirdly, it was perceived that teaching C would provide a foundation for moving towards object-oriented techniques if and when the need arises in the future. Pascal was chosen for the undergraduate variant of the course, firstly because it satisfies the above requirements, and secondly, because it provides a stepping stone towards C.

The applications lectures were intended to provide a broad introduction to the industrial use of computing technology which students were unlikely to receive elsewhere in their studies. It was considered that these lectures provided a prime opportunity to demonstrate generic themes and trends in software and hardware, and their influence on engineering activities.

Assessment

The course was assessed through coursework, with typical examples given in Appendix 1, and examination, with typical questions given in Appendix 2, both being equally weighted. The former was intended to assess students' ability to assimilate and apply the taught software engineering and programming techniques. The examination was intended to test students' understanding of this material without the safety net of achieving over several weeks on a trial-and-error basis. It was also considered to be a convenient method of assessing the taught computing technology applications material. Within the coursework element, two assignments were set, each designed to assess the students' ability to analyse a given problem using software engineering techniques, and then to translate the problem solution into a functional computer program. Each piece of coursework was to be performed by students on an individual basis. All tutorial classes within the course were centred around the fulfilment of coursework objectives (Appendix 3).

INITIAL RESULTS

Feedback from students, received via objective discussions, prompts for improvements, and a questionnaire, indicating that the course was generally well received (Appendix 4). For example, it is considered that the problems that students encountered stemmed, primarily, from not being familiar with the programming language and lack of experience in using software engineering. However, it was found that students spent disproportionately large amounts of time debugging their programs. Due to this, it is doubtful whether the students fully appreciated the wider usefulness of software engineering techniques, beyond program structuring, at the time of delivery. In fact some students found it to be an encumbrance to their programming effort.

The coursework was carried out to a generally high standard, probably more so than the students appreciated. The measure of success that students placed on their own work was the development of a fully functioning program; anything less was deemed to be a failure. This was somewhat disappointing as, in the marking scheme, emphasis was placed on a relatively small percentage of the marks being awarded for this, the majority being given for the analysis and design of the system and consistent program structure. This view was particularly noticeable with regards to the first piece of coursework, where staff and students concurred that software engineering was indeed a sledgehammer to crack a rather small programming nut.

This problem was not so severe in the second piece of coursework, where a larger problem obviously needed structuring and software engineering was agreed to be a useful tool to achieve this. Also, more programming skills and software engineering experience were acquired by this time and students therefore performed somewhat better than on the first coursework assignment. However, fewer working programs were submitted!

One major problem facing the staff was the volume of marking, particularly of the first coursework assignment, when there was a great pressure to return it to the students before they embarked on the second. However, the C and Pascal marking was made more straightforward and more readily quantifiable by the well-structured nature of the code produced by all students.

Two other, perhaps more severe, problems were caused by the fact that only 20 personal computers were available within the department at the time for teaching. Particularly in the undergraduate variant of the course, this made it necessary to have otherwise unnecessarily small groups in tutorial sessions. This slowed the delivery of the course, causing frustration amongst many students. This delay also reduced the coverage of computing technology applications to just a single lecture on each, whereas it was originally envisaged that these would be complimented by practical sessions.

Overall, however, the first year was viewed as a success. Not only did the students carry out a similar amount of programming work as in previous years, but additional material was also covered in the time previously taken for programming. From the viewpoint of the M.Sc. students, their variant of the course was relatively demanding, since they had to perform each item of coursework within the Autumn term only. However, despite this, the quality of the work that they produced was at least as good as that produced by the undergraduates.

EVALUATING THE COURSE

Based on the data given in the appendices the course is considered to be a good initial effort. Replies to the questionnaire, Appendix 4, indicate that the students appreciated the value of the course for its relevance and its challenging nature. From Fig. A1, it is clear that students made progress in their understanding and ability to apply the taught software engineering and programming techniques in their coursework.

In comparison to the previously discussed objectives, it is considered that a high degree of achievement has been attained on each point. However, further developments, several of them based on feedback from the students, are possible and desirable:

- Students expressed a preference for a greater number of small but complete examples that take them through both software engineering and programming.
- The current coursework tasks highlighted the initial difficulties that students had in understanding and applying systems analysis techniques. This may have resulted from the topic of coursework #1, Appendix 1, being sufficiently small and familiar to students for them to make good progress without the need for software engineering. *Ad hoc* comments from undergraduate students also indicated that they considered coursework #2 to span too great a part of the second term.
- In teaching Pascal, the undergraduate variant of the course does not serve students' future requirements for programming skills as well as it might. This view is reinforced by the awareness of several undergraduates with some previous programming experience.
- A larger portfolio of coursework topics is required to ensure that each new delivery of the course does not result in the reworking of the 'standard' problem.
- Examination performance in the area of programming is also of concern (Appendix 2).
- Many students indicated that they would like more time on the course (Appendix 4). The reasons for this are twofold—some finding the work very taxing, whilst others demonstrated considerable enthusiasm and competence.

DISCUSSION

From the experience gained to date, three major themes can now be considered: short-term improvements, the ways in which the course enables spin-off activities and the longer-term future for the course.

Course developments in the subsequent year

Changes in both the available computing resource and in the structure of the M.Sc. course have significantly influenced recent developments. The introduction of 40 networked personal computers within one room has overcome the need to divide students into detrimentally small groups for programming tutorials. This has also allowed time to include hands-on sessions for various applications. Unfortunately this change has only been beneficial on the undergraduate variant of the course since, as a result of modularization, the contact time for Computing Technology within the M.Sc. syllabus has been reduced by 25%. This restructuring has also forced the exclusion of coursework #1 for the M.Sc. students.

For the undergraduate students, the first coursework assignment has been modified to exclude the programming task. Students are now given well-structured code for the quadratic problem and required to generate the associated software engineering by reverse engineering the code. It is anticipated that this approach will promote deeper understanding of software structure and statements, the taught software engineering technique and the relationships between the two. Also, the subject of the second coursework assignment has been extended to allow small group work, with each individual student being made responsible for the software engineering and programming of part of the overall system. The aim of this is to additionally demonstrate the benefits of software engineering as a tool for communication and understanding between system developers.

Spin-off activities

In parallel with the first implementation of the Computing Technology course, a series of open-ended 'mini-projects' were introduced for first- and second-year undergraduate students as a more challenging and interesting alternative to a conventional laboratory programme. For the first-year students, the first of these mini-projects provided an introduction to the computing facilities within the department. An important objective of this exercise was to ensure that the subsequent teaching of programming languages was not clouded or slowed by initial system familiarity problems. The Computing Technology course was seen as a focal point in justifying the need for this mini-project, with more general familiarity being an additional bonus.

Stemming from the teaching of software engineering, second-year students who participated in a mini-project concerning the design of a plastics database were given brief tuition in the creation of data flow diagrams and data dictionaries, which they used for system design and as a means of communicating their findings.

Considering the M.Sc. variant of the course, one student made substantial use of the taught techniques in his dissertation, which concerned the engineering and programming of a finite element program for thermal analysis [5]. Considering the relatively short teaching time prior to commencement, this was an ambitious project. However, the results were impressive and a clear indication of the benefits of the teaching regime.

Since introducing the course, the level of awareness on the part of those academic staff and students within the department who are not directly involved with the course has grown rapidly. Key features of the current academic year that have resulted from this are:

- The initial mini-project for first-year undergraduates has been extended into an assessed exercise that involves in-depth use of word-processing, graphical and spreadsheet software, integrating the results to produce a final report on some selected topic. A significant factor that has enabled this has been the increased number of computers available specifically for teaching.
- Several final-year undergraduate students have either adopted the software engineering technique as an aid to documentation or are carrying out projects where software engineering is a central theme. This is considered to be a significant development since such students have not had the benefit of formal tuition in this topic.

Future developments

The suggestions from students for a number of complete but small examples that take them through both software engineering and programming has merit, since such examples would reinforce three important themes: the interaction between different elements of the software engineering technique, the transition from software engineering to programming, and the nature of good program structure and the way that it should be dictated by the software engineering solution.

If the above examples could be presented prior to or in parallel with the first piece of coursework, students' initial difficulties, exemplified in Appendix 1, may be mitigated. Also, subdivision of coursework #2 into discrete software engineering and programming components, where the former is assessed prior to students commencing the latter, would help to reinforce the philosophy of these elements of the course more strongly.

It is considered by members of staff that a move from Pascal to C would be beneficial to the undergraduate variant of the course. Firstly, this would make the undergraduate and postgraduate variants more consistent. Secondly, it would make the undergraduate variant more relevant to industrial need. Thirdly, it would better place students to carry out subsequent projects related to various research activities within the department, and provide a more sympathetic basis for programming activities in other courses within the degree syllabi. Were C to be introduced in the next academic year, it would be done with the benefit of two years' delivery on the postgraduate variant, as discussed at the end of this section.

Considerable scope exists to introduce alternative coursework topics and/or alternative approaches to the existing topics. Numerous topics can be generated, such as graphics systems, user interfaces, databases, and small numerical analysis and statistical tasks. Also, alternative approaches to a given problem can be considered, e.g. given the software engineering and source code for an existing system, along with a revised set of functional requirements, students could then carry out the necessary re-engineering and programming to implement an updated system.

Changes to the assessment procedure are also being considered. Primarily, this concerns the need for an examination. Appendix 2 indicates that only a small proportion of students attempted questions that tested their understanding of programming. Regardless of this being a result either of their lack of understanding, or examination being an inappropriate assessment mechanism, then it may be appropriate to introduce an element of continual assessment during the course and remove programming as an examinable topic. If this were done, continual assessment could also include software engineering. Also, provided that students' assimilation of the material in the applications lectures could be tested via a relatively small essay coursework, then the entire course could be assessed on a continuous basis. This is currently receiving serious consideration.

Staff are investigating how students' understanding gained from this course can be integrated elsewhere in the department's degree course. The wide ranging opportunities that are currently available are indicated in Fig. 2, which shows those topics taught at undergraduate level within the department in which a system analysis approach can be adopted, either as a focal point or as a supporting view.

In addition to contributing to existing topics, the existence of the Computing Technology course creates opportunities for the development of new courses. An example of this is the possibility of introducing a second-year course that addresses the analysis and design of concurrent processing systems for manufacturing process control. In addition to this being a significant end in itself, this could also create a foundation for a future final-year subject that integrates the required techniques and technologies for concurrent or simultaneous engineering.

In addition to allowing the development of a more streamlined Computing Technology course, the recent increase in computing facilities now provides an opportunity for using computer-aided software engineering. If this were to be adopted, two major benefits would be expected. Firstly, it would allow students to focus more strongly on the problem to be solved, rather than ensuring adherence to the many rules within the technique. Secondly, it would allow staff to concentrate on assessing students' understanding of how to apply the taught material, rather than their correct implementation of the previously mentioned rules.

Considering the M.Sc. course in particular, it is

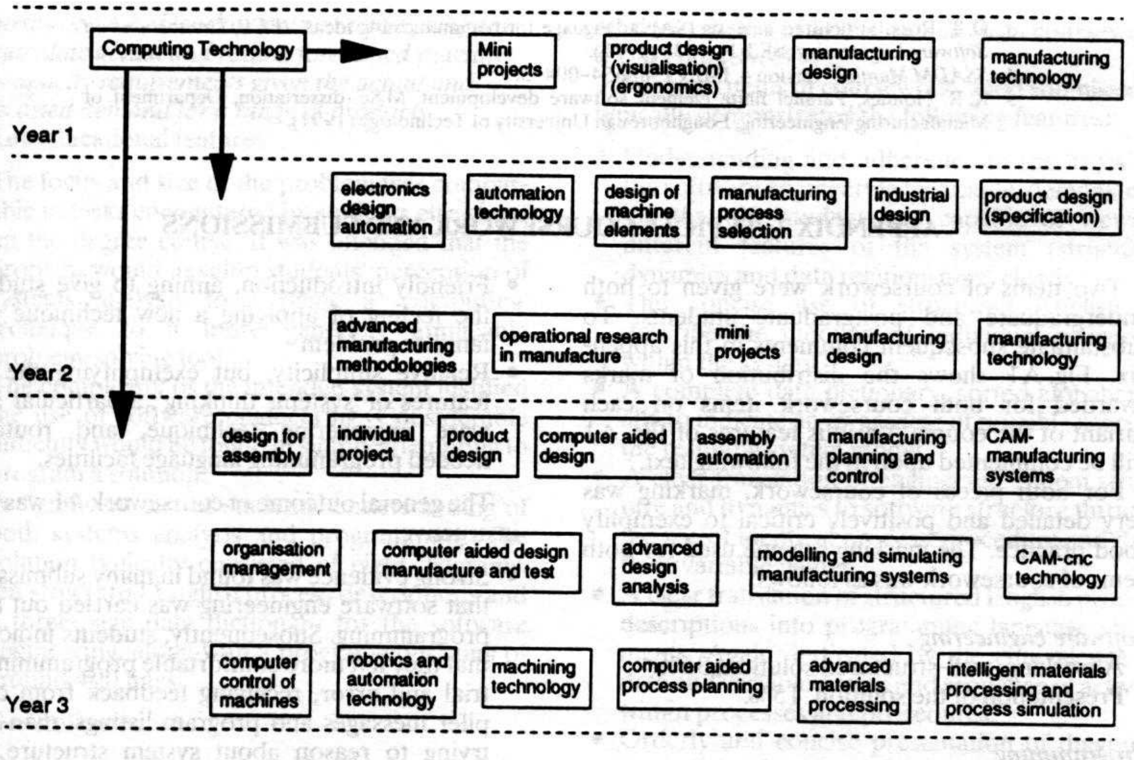
Computing Technology
Mini projects
product design (visualisation) (ergonomics)
manufacturing design
manufacturing technology
Year 1
electronics design automation
automation technology
design of machine elements
manufacturing process selection
industrial design
product design (specification)
advanced manufacturing methodologies
operational research in manufacture
mini projects
manufacturing design
manufacturing technology
Year 2
design for assembly
individual project
product design
computer aided design
assembly automation
manufacturing planning and control
CAM-manufacturing systems
organisation management
computer aided design manufacture and test
advanced design analysis
modelling and simulating manufacturing systems
CAM-cnc technology
computer control of machines
robotics and automation technology
machining technology
computer aided process planning
advanced materials processing
intelligent materials processing and process simulation
Year 3

Fig. 2. Current undergraduate topics where the techniques taught in the Computing Technology course may be adopted.

now recognized by the staff involved that teaching C on a need-to-know basis, focused by the needs of a software engineering solution, provides more stimulus and rapid understanding of specific language facilities than teaching C in isolation. However, a problem with this approach is the risk of not giving sufficient emphasis to the basic and/or generic aspects of the language. Thus, there is a need for a mixture of both approaches, with a bias towards the former. This could be achieved via occasional 'let's spot generic issues' sessions during language teaching.

A more significant problem, however, is the recent modularization of the M.Sc. programme, which has created the need to revise substantially the structure of this variant of the course. The outcome of teaching the course under this regime is currently being examined and it is anticipated that further revisions may take place prior to the 1993/94 academic year.

In the longer term, it is considered that the development of distance learning facilities and the possible introduction of intensively taught modules will provide further opportunities for a new and more appropriate Computing Technology course structure on the M.Sc. programme.

CONCLUSIONS

The Computing Technology course discussed here adds considerably more value to the students' education than the traditional programming course which it replaces, specifically:

- In addition to learning how to program, students also learn about systems analysis and the use of industrial software products. The time taken to achieve all of this is the same as that required previously for programming only.
- Many new opportunities for project-based work have been created, not only for those projects having a central computing theme, but also in allowing students to adopt a more structured approach to projects in other areas.
- Opportunities have been generated for alternative approaches to the teaching of other existing undergraduate subjects and for the introduction of new subjects from a systems viewpoint.

Student feedback has been both informative and encouraging in helping to ensure that the course meets its original objectives to an even greater extent in the future.

REFERENCES

1. R. L. Wood and L. E. Davis, Software engineering for manufacturing engineers. IEE colloquium on 'The teaching of software engineering—progress reports', IEE digest no. 1991/159, pp. 3/1–3/5 (1991).
2. E. Yourdon, *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, NJ (1989).

3. D. T. Ross, Structured analysis (SA): a language for communicating ideas, *IEEE Transactions on Software Engineering*, SE3(1), 1633 (1977).
4. *SSADM Manual*, version 4, ISBN 1-85554-004-5.
5. K. R. Holmes, Parallel finite element software development. M.Sc. dissertation, Department of Manufacturing Engineering, Loughborough University of Technology (1991).

APPENDIX 1: TYPICAL COURSEWORK AND SUBMISSIONS

Two items of coursework were given to both undergraduate and postgraduate students. To substantiate subsequent comments in this appendix, Fig. A1 shows the distribution of marks awarded for both coursework items on each variant of the course. Various features of Fig. A1 will be commented upon in the following text.

For both pieces of coursework, marking was very detailed and positively critical to exemplify good practice. The marking scheme used for both items of coursework was as follows:

Software engineering

A working, well-structured solution, 35%.
Presentation of the solution, 15%.

Programming

Software that meets the objectives and whose structure is consistent with the software engineering solution, 35%.
Presentation to the user and the programmer (i.e. the user interface and in-line source code documentation) with evidence to demonstrate successful software operation, 15%.

Coursework # 1: develop an interactive program to solve a quadratic equation

Key educational features:

- Historical precedent extended through the need for a user interface that allowed repeated execution.
- Friendly introduction, aiming to give students the feeling of applying a new technique to a familiar problem
- Relative simplicity, but exemplifying the key features of 'systems thinking', a particular software engineering technique, and routinely needed programming language facilities.

The general outcome of coursework # 1 was very instructive:

- Strong evidence was found in many submissions that software engineering was carried out after programming. Subsequently, students indicated that they felt more comfortable programming by trial and error, receiving feedback from compiler messages and program listings, than with trying to reason about system structure and dynamics via diagrams.
- A small group of students misused the software engineering technique because they confused it with their previous experience of drawing flowcharts. This need to 'un-learn' a previous approach is seen to be a significant problem since, to promote such students' confidence, their previous approach must be dissected and demonstrated to be less informative.
- As can be seen from Fig. A1, the marks awarded for coursework # 1 were relatively low and tightly grouped.

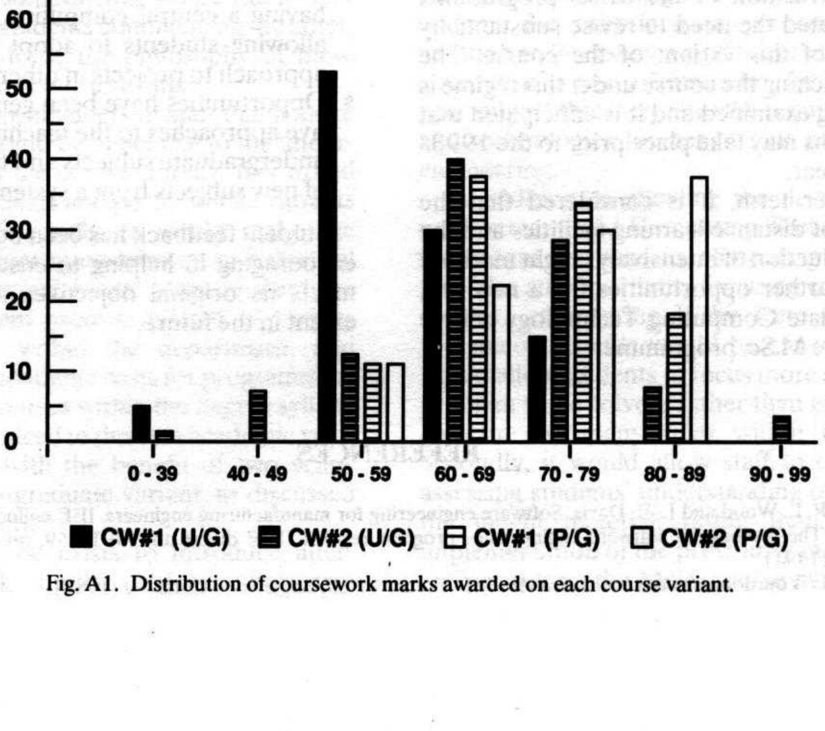

Fig. A1. Distribution of coursework marks awarded on each course variant.

Coursework # 2: Develop an interactive program to calculate actual and rolling forecasted machining capacity requirements given the actual and forecasted demand for a range of products

Key educational features:

- The focus and size of the problem was comparable to tasks encountered by students elsewhere on the degree course. It was intended that the problem would develop students' perception of systems analysis from being a 'computing' technique to a more generally applicable problem-solving tool.
- The complexity of the intended system justified the use of software engineering. Considerable difficulty would be found in simply attempting to program a solution.
- A significant test of students' understanding of both systems analysis and programming. The solution typically consisted of eight diagrams, ten structured English process descriptions and a three-page data dictionary for the software engineering, along with a program consisting of around 400 lines.

General outcome:

- Despite the relative size and complexity of the task, both undergraduate and postgraduate students demonstrated a distinct improvement in their understanding of both software engineering and programming, as can be seen from Fig. A1.

Constrasting students' performance on each coursework activity, from Fig. A1 it can be seen that for coursework #1, 56% of the undergraduates scored less than 60%. For coursework #2, the proportion of students within this range decreased to 21%. Also, 11% of the undergraduates scored over 80% for coursework #2, whereas no undergraduates attained marks in this range for coursework #1.

Postgraduate students demonstrated a more consistent performance across the two exercises. However, one notable feature of the postgraduate scores is that the number of students attaining marks in excess of 80% doubled from coursework #1 to coursework #2.

For both items of coursework, good submissions typically demonstrated the following features:

- Understanding and adherence to the 'rules' of the software engineering technique, demonstrating that the student was capable of viewing different features of the system (structure, dynamics and data relationships) clearly.
- The concise use of structured English to describe the function of individual processes within the system.
- A complete data dictionary, sorted alphabetically to indicate an awareness of possible users of the resulting system model.
- A clear translation of engineered system structure and dynamics to software structure through the use of identical process/procedure and data flow/variable names.
- A clear translation of structured English process descriptions into programming language statements. Again, indicated by the equivalence of data and variable names and sequence of activity within processes and procedures.
- Orderly and concise presentation of diagrams and text within the software engineering part, along with a listing of the resulting program.
- Inclusion of exemplary software input and output, demonstrating that the resultant software functions correctly.

In contrast, poor submissions displayed features such as:

- Inconsistencies between diagrams for the structure and dynamics of the system.
- Inconsistencies between diagrams depicting different levels of detail in system structure.
- An incomplete and/or ambiguous data dictionary.
- Program structure that does not match the associated software engineering.
- A written description of what the student thought they should do, but did not, to carry out the coursework task.

APPENDIX 2: TYPICAL EXAMINATION QUESTIONS

In addition to assessing the students' appreciation of the industrial impact of computing technology, the examination was intended to test their resultant understanding of the taught techniques, rather than their ability to apply it over a period of several weeks.

Typical questions in each area of the course were as follows.

Software engineering

Using the example of calculating the average of a set of numbers, illustrate the use of event lists, context diagrams, data dictionaries, state transition diagrams, data flow diagrams, mini-specifications and structure charts. Assume that input to the system from the user is always correct and does not need to be validated.

Programming

1. Explain the effect of using each of the following storage classes (in ANSI C) automatic, static, external and external static. Illustrate the concept of scope with an example piece of code.
2. Given the following statement, what does the function tell you about the intention of the programmer?

   static int forecast;
   int value_ct(const int arr[], int value, int n)

Will replacing *int* value and *int n* with *const int* value and *const int n* enhance the protection of values in the calling program?

What effect will this substitution have on the function?

Computing technology applications

Discuss the enabling role of computing technology in product design, highlighting the different technologies that have evolved to support the creative and engineering science aspects of design.

Students' examination performance was typical of, if not a little better than, their examination performance in other subjects. However, it was not compulsory to answer a question of each of the above types. As a result, 87% of the students answered software engineering questions, but only 29% answered the programming questions. For those answering the software engineering questions, the spread of marks was similar to that for coursework #2 (Fig. A1). For the programming question, 50% of the students who answered them gained less than 50% of the available marks and 30% gained in excess of 70%. The different numbers of students attempting each type of question clearly indicates that they were less confident in their programming knowledge than their knowledge of software engineering. For those who attempted the programming questions, there was a clear split between good and poor achievers. This raises two questions: 'Is programming so difficult that few students actually understand, but achieve in coursework through trial and error?', and 'Is an examination the most appropriate method of testing students' understanding of programming?' Debate continues within the department on the answers to these questions.

APPENDIX 3: TYPICAL CONTENT OF PROBLEM-SOLVING CLASSES

Whilst problem-solving classes were strongly geared to supporting the coursework activities, such classes for software engineering and programming progressed along slightly different lines. For tutorials, both undergraduate and postgraduate classes were divided into groups of around 15 students.

For each coursework item, every group received an initial software engineering tutorial which encouraged the development of a solution through interaction within the group. These tutorials were open ended, directed by prompts, ideas and suggestions offered by students. Progress towards the solution was dictated by the pace and focus of the group, typically achieving around 20–25% of the required software engineering solution within a 2 hour tutorial. Subsequent software engineering tutorials were run as *ad hoc* sessions where a member of staff responded to problems on an individual student basis.

Programming tutorials were staggered with those for software engineering so that the former may provide an initial focus for the latter. Students were encouraged to appreciate the need for particular classes of program functionality, leading to the need for particular types of statements and supporting data definitions. Prime examples of this are the need for input and output, and for error checking to be included in the former when the software interacts with human users.

APPENDIX 4: STUDENT FEEDBACK

Feedback was obtained via objective discussions, including prompts for improvements and a questionnaire.

Objective discussions and prompts for improvements

Consistent comments were obtained from both undergraduate and postgraduate students. The general points were as follows:

- A desire for more time overall (some found the course very demanding and wanted more staff input, others found it especially interesting and wanted to do more).
- A desire for more tutorials (complete examples of software engineering and programming, more opportunity for supervised programming, more tasks that focus on specific features of software engineering and the language)
- A suggestion that the major piece of coursework be divided into two discrete sections—software engineering followed by programming.
- A suggestion from several undergraduates with previous programming experience that Pascal should be replaced by a more industrially relevant language, such as C.
- The need for better printing facilities (introduction of the computing technology course placed an unexpected demand on printing).
- A specific comment from postgraduate students was that there should be no examination questions on the C programming language (they did not see the point of testing their understanding rather than their use of C).

A range of less specific comments implied that students were looking for stronger guidance and more frequent assessment to make them work at a more measured rate.

The questionnaire

Table A1 summarizes the questions put to the students, indicating the proportions of 'yes' and 'don't know' replies. Where only one type of reply is given to each question, the balance is implied to be 'no'.

Considering each of the questions in turn, from Table A1 it can be seen that there was a marked difference between the previous programming and software engineering experience of the undergraduate and postgraduate students. This is also thought to be an influence on replies to subsequent questions.

There was a marked difference between undergraduate and postgraduate replies as to the usefulness of the taught programming language. This probably results from (i) greater programming experience amongst the postgraduates, and (ii) limited awareness amongst the undergraduates of alternative languages to Pascal.

Surprisingly, a similar proportion of undergraduates and postgraduates had some prior software engineering experience.

When asked about the more general relevance of software engineering, a surprisingly low proportion of postgraduates indicated their belief of its relevance. On both variants of the course, scope exists for communicating the general applicability of software engineering and systems analysis more clearly.

Considering whether software engineering assisted programming in the coursework, undergraduates and postgraduates replied in similar proportions. The relatively low proportion of students that considered software engineering to be of benefit could be based on one or both of the following. Firstly, students may have perceived that the software engineering forced them to think in what they considered to be an overly complex way. Secondly, their use of software engineering may have forced students to use more sophisticated programming language facilities that they would otherwise not have used.

Considering the influence of software engineering on the overall quality of the resulting software, a higher proportion of postgraduates appreciated the benefits of software engineering. This is probably based on their greater experience of both software engineering and programming, allowing them to appreciate better the importance of software structure. A large proportion of undergraduates may not be sufficiently experienced, even after the course, to recognize the characteristics of 'better' programs.

The relatively small proportion of both undergraduates and postgraduates who thought that a reasonable balance between lectures and tutorial had been achieved indicated that both groups thought that more complete examples would be of significant benefit.

Considering the appropriateness of the assessment procedure, it was found that 19% of the undergraduates and 26% of the postgraduates wanted 100% coursework, whilst 21% of the postgraduates wanted no coursework. No undergraduates indicated that they would prefer the latter. However, several suggestions were made as to alternative weightings between coursework and examination.

Regarding the two questions concerning coursework relevance. The general view was that the topics were relevant but, perhaps, more appropriate ones could be identified.

Considering the overall worth of the course and

Table A1. Questions presented to the students and their replies

Question	Undergraduate response (%)	Postgraduate response (%)
Did you have prior programming experience?	yes = 48	yes = 100
Will the taught programming language be of future use to you?	yes = 52 don't know = 29	yes = 95
Did you have prior experience of software engineering?	yes = 24	yes = 32
Is software engineering relevant to other engineering activities?	yes = 72 don't know = 14	yes = 68 don't know = 32
Do you think that software engineering helped programming in the coursework?	yes = 52 don't know = 14	yes = 68 don't know = 16
Do you think that you obtained better programs by doing software engineering?	yes = 38 don't know = 10	yes = 74 don't know = 5
Did you receive a good balance between lectures and tutorials?	yes = 14	yes = 37
Do you think the balance between coursework and examination was appropriate?	yes = 62 don't know = 19	yes = 53
Were the coursework topics relevant to the degree course?	yes = 71 don't know = 5	yes = 100
Were the coursework topics relevant to the Computing Technology course?	yes = 100	yes = 100
Was the course worth doing?	yes = 90	yes = 79
Do you think that Computing Technology has a place within the degree syllabuses?	yes = 95 don't know = 5	yes = 95 don't know = 5

its place within the degree syllabuses, the general consensus was that they received a worthwhile educational experience for their efforts. To substantiate their replies, many students commented on the importance of computing technology in manufacture and, despite the difficulty of the course, they derived satisfaction from it.

Lesley E. Davis is a lecturer in the Department of Manufacturing Engineering at Loughborough University of Technology. She graduated from Royal Holloway College, London with a B.Sc. (Hons) in mathematics and from the University of Birmingham with a M.Sc. (Eng.) in operational research, she also holds a postgraduate certificate of education. She is a member of the Operational Research Society and has current research interests in the fields of simulation and decision aiding.

Stephen T. Newman graduated from the University of Aston in 1982 in production technology and production management. In 1990 he gained his Ph.D. at Loughborough in the simulation and flexible machining cells. He is currently involved in both Eureka and national research programmes in flexible machining cell design with special interests in tool management.

Kevin R. Holmes graduated from the University of Newcastle upon Tyne in mechanical engineering in 1987. He graduated with distinction, in 1992, from the M.Sc. Computer Integrated Manufacturing course, run by the Department of Manufacturing Engineering at Loughborough University of Technology.

Robert L. Wood graduated in mechanical engineering in 1980 and is a Chartered Engineer and a member of the British Computer Society. He became a lecturer in the Manufacturing Engineering Department in 1989, after almost 10 years in industry. His research interests centre on the use of numerical methods and software engineering in the simulation of manufacturing processes.