Int. J. Engng Ed. Vol. 12, No. 4, p. 286–298, 1996
Printed in Great Britain.

0949-149X/91 $3.00 + 0.00
© 1996 TEMPUS Publications.

Software Design for Engineers: A Practical Undergraduate Course in Structured Design*

J. E. COOLING
Department of Electronic and Electrical Engineering, Loughborough University, Loughborough LE11 3TU, UK

This paper describes the structure, development and implementation of an undergraduate course in structured design methods for software systems. It is designed primarily for second year engineering students, to provide them with design skills for the development of real-time systems. The course is a highly practical one, assessment being entirely by coursework. It aims to allow students to gain experience, both individually and in group working. The practical work is underpinned by the use of a modern PC-based CASE tool. Cross-working is an integral part of the scheme, as also is the peer marking of coursework.

INTRODUCTION

Twenty-five years ago few control, monitoring, data-logging or display systems contained digital computers. Software engineering did not exist as a topic within the undergraduate curriculum. Most engineers viewed computers as a tool to solve mathematical problems, simulate systems, evaluate control loops, etc. To use these (mostly mainframe) computers, the only skill required was that of programming.

All this began to change with the arrival of the microprocessor. Initially, owing to the high costs, its penetration into engineering systems was relatively slow. However, by the mid-1970s, when the prices had dropped substantially, it began to be used in large numbers. This created a great demand for programmers and proficient ones at that (with relatively small storage and modest clock speeds, the micro required efficient programming). Thus, for the best part of the next decade—even today in some organizations—software development for microprocessors was seen essentially as a programming task.

During this period, software engineering was establishing itself as a subject in its own right. It embraced a whole range of software topics: analysis, design, modelling, testing, metrics, configuration management, etc. Yet these made little impact on the microprocessor field—many companies continued to develop software in a most primitive fashion.

During the mid- to late 1980s, concern was expressed about the quality of such software-based systems. It was seen that, to improve the long-term situation, better training and education was required. As a result the IEE and the BCS advocated a formal approach to the education of software engineers [1].

It was also clear to us at Loughborough that software engineering should form an important part of engineering courses. We began to introduce the topic—as a subject in its own right—in the late 1980s [2]. By 1991 it was being taught in all 3 years of our undergraduate degree course. For a variety of reasons these had been developed in a somewhat *ad hoc* fashion. With hindsight, the result of this was predictable. Each year, taken in isolation, was satisfactory. Unfortunately, considered as a whole, the 3 years were poorly integrated and lacked cohesion.

To remedy this, a major course review was carried out, resulting in a new syllabus and structure. The aim was to provide our engineering undergraduates with a sound foundation in software engineering. At present, the revamped course is now into its second year of operation. Year 1—which remains essentially unchanged from the earlier days—concentrates on program design. Year 2 introduces the students to software design, with special reference to embedded applications [3]. It has a very high practical content, the design methodology being based on structured design principles [4].

The rest of this paper is given over to describing in detail the background, detail and operation of this second year course.

SETTING THE SCENE—COURSE CONTEXT AND OBJECTIVES

The course in context

In order to understand the rationale of this course some explanation about its background is

*Accepted 26 March 1995.

necessary. First, it must be placed in context. We are not dealing with software engineering *per se*. Rather, it is the software aspects of an engineering course. Second, the amount of time available within the syllabus is quite limited. Third, it needs to build on first year material—and integrate with other second year modules.

Within engineering, a (if not the) major area of work is that of real-time embedded systems. Further, safety-critical aspects are becoming more of an issue in such computer-controlled applications [5]. Ideally, the syllabus would be a very wide one, covering all the relevant topics in embedded software development. There is, though, relatively little time available for the software work; one unit of 36 hr. Fortunately this is not quite as limiting as it seems, as second year students also study (amongst other subjects) microprocessor engineering, digital electronics, computer-aided electronic engineering (CAEE) and computer peripheral systems (optional).

All the students have met software engineering in the first year. There the emphasis is on program and algorithm design, the teaching language being Modula-2 [2]. The course involves both individual and group working, developing software in a PC environment. Hence, entrants to the second year are generally quite proficient in terms of program development.

Fundamental philosophy

From the beginning it was clear that the time available for the course would heavily influence its structure. It was felt that, to achieve useful results, the approach taken should be a focused one. Moreover, the feedback from graduating students indicated that our existing course was weak in one particular area: design aspects. We therefore decided on the following.

1. To provide a sound if somewhat limited theoretical and practical course on software design.
2. To direct it at the area of real-time embedded systems.
3. To achieve the desired results by having students 'learn by doing'.
4. To allow the students to develop design skills, both individually and in groups.
5. To make the coursework reflect the nature of software development within the industrial and commercial worlds.
6. To present the subject as being interesting and useful (and not merely another academic hurdle in the race to a degree).
7. To introduce CASE tools.

One major decision we faced was 'which design method?'. In broad terms the choice lay between object-oriented (OO) and structured methods. This is a contentious subject [6,7], clouded by the excess of hype relating to OO techniques. We felt, in the light of our own (and other) work [8–10], uneasy with the introduction of OO design. Moreover, it could also pose problems from the language point of view.

In contrast, structured methods are extremely widely used in the real-time field, particularly in embedded applications. It is an established and proven method, with extensive CASE tool support. For instance, it has been the preferred design method of Motorola for some years now [11], it is used by British Rail Research as a preferred design technique for safety-critical signalling systems [12] and it is recommended for use in the motor industry in the first set of software guidelines produced by MISRA [13].

Consequently, the following was decided.

1. The course would be based on structured design methods.
2. Design would be supported using CASE tool technology.
3. All assessment would be done by coursework, with no written examinations.
4. The students would be required to develop software in two stages. Stage 1 would be done individually, stage 2 being a group activity. The objective of the second stage would be develop software for a complete (if small) realistic system.
5. A major factor in this would be the requirement to implement the work of other designers.
6. They (the students) would play an active part is assessing the software designs.

A final—difficult—issue was one relating to safety-critical systems. In such applications an important topic is that of formal specification methods [14]. We felt it important to include these in the syllabus, demonstrating their application to real software. The challenge was to do this is a limited amount of time. A twofold solution was devised. First, a set of lectures would be given over to the basics of formal methods (illustrated using the Vienna development method (VDM)) [15]. Second, the students would be required to use this information as part of their individual design task.

Within the university, the courses are organized on a modular basis. To align with this, the following overall course structure was defined: total time timetabled, 36 hr; lectures, 20 hr over two terms; practical (timetabled hours), 16 hr over two terms; and practical (own time—estimated), 75 hr over 2.4 terms. It also incorporates some quite new ideas (for us, that is). This includes a combination of software module development on an individual basis, group design tasks (which includes the assessment and use of a set of individual software modules) and group implementation tasks (which, in part, assesses and uses the results of group designs).

During group design, each group is required to use the modules produced by other students (not their own). During implementation, each group is required to implement the design of another group. This is a similar idea to that used in the Crossover project [16]. However, the assessment is a true peer one—the students mark other students work. Moreover, the marks awarded form part of the total allocated for the course module.

The taught content

The conventional approach to software development is a top-down one [17]. That is, start at the requirements level and finish with the source code. This is fine and reasonable. Or so it seems. Unfortunately, in practice, this style creates a number of difficulties for the novice designer. One major problem is that it requires the ability to take a broad (somewhat abstract) view of system issues initially. Newcomers to design have insufficient experience to take this viewpoint—things need to be more concrete.

This course turns the conventional method on its head by adopting a bottom-up approach. It starts with what the student knows at the start of the course, then builds on that knowledge, step by step (Table 1). The gap between the steps is kept as small as possible. Many 'mental' hooks are provided to bridge the gaps and make the learning process a straightforward one. Moreover, the reasons for and advantages of using several design stages are made clear. The relationship between the various stages is well defined, resulting in an integrated design approach. In parallel with this, the students study the fundamentals of formal specification techniques (Table 2).

The practical content

There are two distinct sections of practical work: assessed and non-assessed. The non-assessed activities are intended to introduce the students to CASE tool technology, allow the students to gain experience with diagramming techniques and directly support the taught material. This takes places during the first term. The students are provided with appropriate material, timetabled laboratory sessions and full laboratory supervision.

Term 2 is given over almost entirely to the assessed (course) work. This coursework is based on developing software for a target embedded system, performed in three stages (Fig. 1). Here a PC is used as the target system.

Task 1. Task 1 is an individual one. Each student is asked to develop one specific software module, to demonstrate an understanding and use of the following.

1. Structured design principles.
2. Source code (program) aspects—the style, layout, naming, commenting, abstraction, information hiding, interfaces, application of pre- and post-conditions and provision of test information.
3. Modularization, with special reference to 'service' modules.
4. Design based on diagramming, specifically Jackson structure charts.
5. Developing software as a set of abstract software machines using structure chart principles.
6. A CASE tool for developing and recording the design.

Task 2. Task 2 is a group one. Each group (four students per group) is required to develop application software for a real-world requirement. The design is to comprise modules developed by other students for task 1 and an application (monitoring, control and display) task which uses these modules. The students are required to demonstrate an understanding and use of the following.

1. Real-time data flow diagrams.
2. Control transformations and associated state transition diagrams.
3. Process specifications and data dictionaries.

They must show how to develop structure charts from DFDs.

The modules produced in task 1 are marked by the students during task 2. Each group is to be given two sets of modules (making eight in all). They are required to mark these and then select the best four for use in their own task 2 design.

Task 3. In task 3, each group is required to implement the design arrived at in task 2—but not for their own work. They are required to work with the design of another group. The implementation should be taken to a fully operational state.

The designs produced during task 2 are marked by the recipient groups. At the end of the implementation phrase, each group must give a pre-

Table 1. Course content—structured design details

Designing software for embedded systems
A pictorial description of programs
The program structure chart
Designing and building software machines
The module structure chart
Introduction to DFDs
Simple input–compute–output systems
Defining systems dynamic behaviour
The state transition diagram
The real-time data flow diagram
Control plus data
Implementing controls
From DFD to code via the structure chart
Concurrent systems
The task model of software

Table 2. Course content—formal specification methods

The engineering of quality software
Introduction
Foundations for correctness proofs
Formulating pre- and post-conditions
Applying correctness proofs
Introduction to VDM
Implementing VDM abstract data types using Modula-2
Designing correct programs

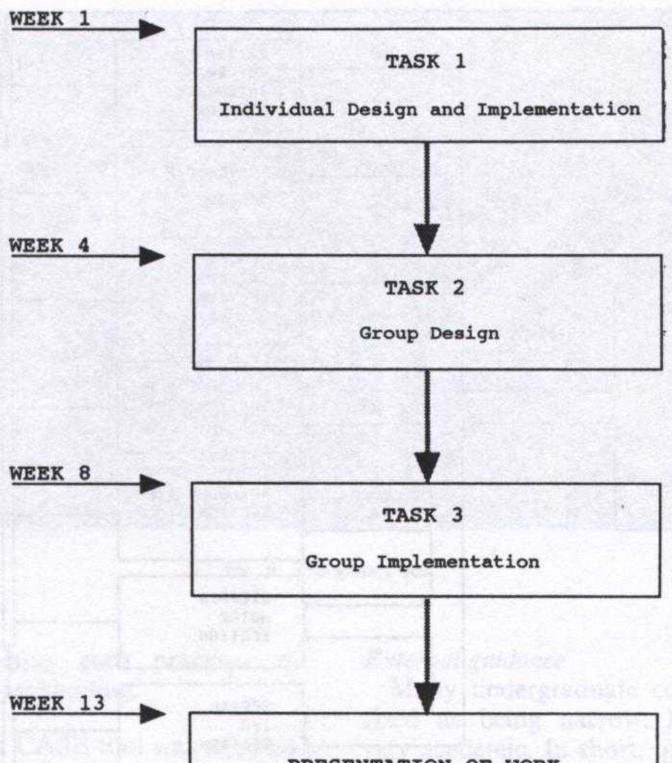

Fig. 1. Coursework structure.

sentation to members of staff. The marks for task 3 are set by members of staff, taking into account all the relevant circumstances.

Mark allocation. Task 1 is allocated 30% of the marks, task 2 40% and task 3 30%, of which 10% is allocated to the presentation.

SUPPORT FACILITIES AND INFRASTRUCTURE

Specialist hardware

To simplify matters, it was decided to use a PC as the target machine. This has the added advantage that the same machine can be used as a host development system. To provide real-world interfaces, a plug-in board was developed for the PC. It houses a variety of circuits, as follows (Fig. 2).

1. Analogue input—single channel, 8 bits.
2. Analogue output—single channel, 8 bits.
3. Switch input—eight switch input lines, opto-isolated.
4. Switch output—eight switch output lines, moderate power capability.
5. Stepper motor drive—small four-phase stepper motor.
6. Serial comms—simplified RS232 interface and standard RS422 interface.
7. External device interface—no specific function.

All the design—circuit and board layout—was carried out in-house. Thirty boards (Fig. 3) were manufactured and installed in the computer laboratory PCs. Sets of test units were also designed and built. Their primary role was to enable the students to test their software functionally.

The final piece of hardware was the item requiring monitoring and control—the 'games box' (Fig. 4). Details of the game operation are given later.

The development environment

To support the coursework, the following facilities and tools are required.

1. Host development hardware.
2. A high-level language compiler.
3. A CASE tool.
4. Word processing and (optional) drawing packages.
5. Draft and high-quality printers.
6. Software for checking out the test units.

The chosen host platform was the PC, for the following reasons. First, all the required facilities can be provided on such machines. Second, the students are familiar with PC systems, being conversant with DOS and Windows. Third, we are able to design and produce our own I/O board for the PC. Fourth, PCs are readily available within the department.

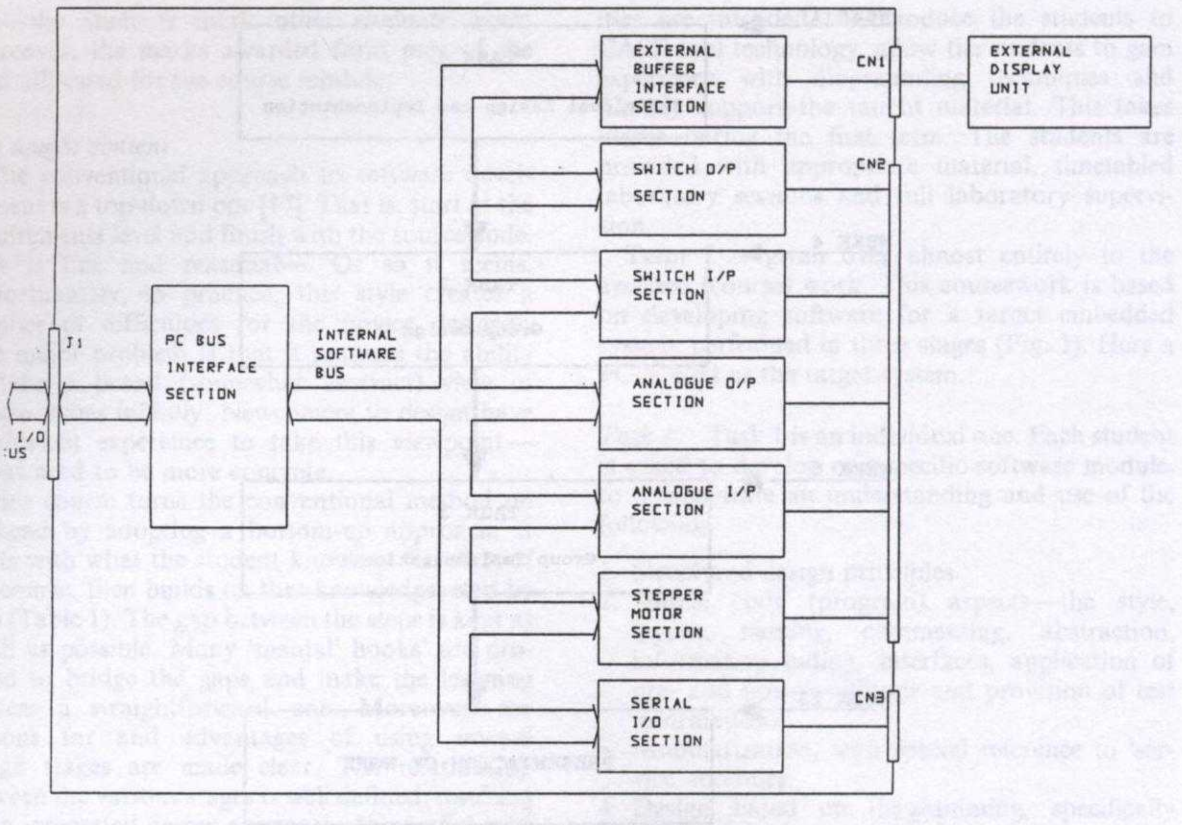

Fig. 2. PC interface board—block diagram.

Fig. 3. The PC interface board and test units.

It was decided that the programming language would, as in the first year, be Modula-2. The chosen compiler (or, more correctly, language environment) was the Stony Brook one. These decisions were made taking into account the following factors.

1. Given the existing time allowance (for the undergraduate course as a whole), it would have been extremely difficult to introduce another language.
2. Modula-2 provides, as standard, most facilities to handle real-world devices.
3. The compiler includes processor-specific facilities.
4. five years experience has showed that this compiler was a solid, trustworthy one.
5. The students were already familiar with the language and the development environment.
6. Modula-2 inherently supports good software engineering practice. We wanted the students

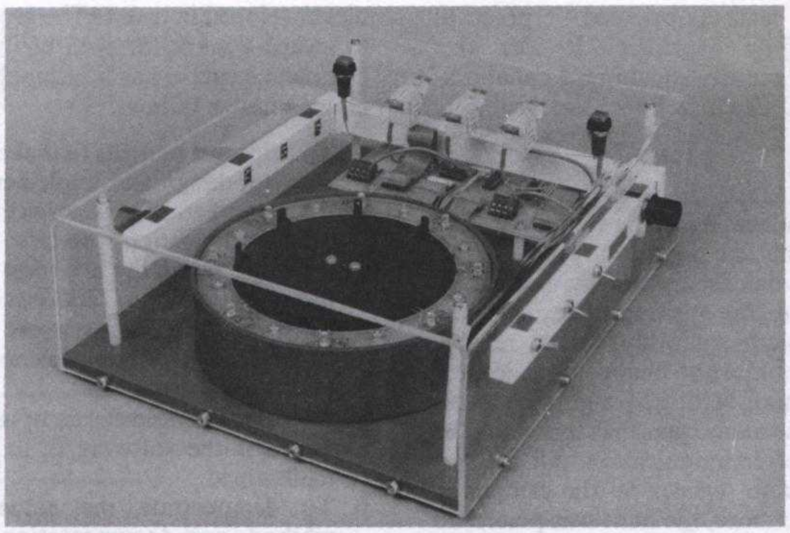
Fig. 4. The games box.

to learn how to apply such practices to embedded systems programming.

A professional PC-based CASE tool was adopted for use in the course, the SELECT Yourdon tool [18]. This provides support for the methods of Yourdon, Ward-Mellor and Hartley-Pirbhai. We had previously used this within research and project work for some years, with very satisfactory results. The most recent version, operating under Windows, was particularly easy to use.

To support the production of high-quality reports, word processing and drawing facilities were provided. The students, however, were not restricted to these particular ones. Any suitable packages could be used. It is also important to provide printing facilities. This, of course, is taken for granted in any modern computer-based course. However, the amount of paperwork produced when using CASE tools is substantial—and leads to high printer workloads. Taken together, these can increase costs significantly (students are notoriously cavalier in their use of 'free' facilities). To deal with this, low-cost printing was provided for draft quality work. However, for high-quality (laser printed) documentation, charges were levied.

Finally, it seemed sensible to provide the students with the means to check out the test units. These devices were generally quite simple and were expected to be highly reliable. However, the failure of even a single unit could lead to a great deal of student time (not to mention demoralizing, frustration and loss of confidence). Furthermore, experience has shown that many students react in a predictable way when faced with malfunctioning systems—they refuse to consider that their software might be at fault. Hence, a suite of programs was supplied. Using these, the students could carry out independent tests of the test equipment. These programs also verified the correct functioning of the PC hardware.

External guidance

Many undergraduate courses have been criticized as being narrow, highly specialized and very academic. In short, quite removed from real engineering. It was important to us that we did not go down such a route. Our objective was to develop a course which addressed the issues of modern practical software engineering (this does not imply diluting its intellectual content). To assist with this, it was decided to involve a senior industrialist as an external adviser to the course, with the following brief.

1. Bring an industrial perspective to the topic.
2. Review and assess the course material.
3. Provide advice concerning coursework topics.
4. Evaluate how well the objectives were met.
5. Guide future refinements and developments of the course.

It is essential that the adviser should have a background and experience which is relevant to the course. A second point is that a person of some seniority is required—but one who is also technically active. We were fortunate to acquire the services of the technical director of a well-known SCADA (supervisory control and data acquisition) company as the course's first industrial adviser.

THE COURSE IN OPERATION

Preliminaries

The overall aspects of the software engineering module were presented to the students at the beginning of the course. We felt it was essential they should have a clear view of its objectives, structure and mode of operation—before travelling down their chosen path. In particular, the rationale for using a peer marking scheme needed to be expounded. Our basic philosophy—that of a student-centred, learning-oriented practical course—was carefully explained. It was also made clear that we actively encouraged collaborative efforts. This, after all, is how most engineers work in real life. However, being realistic, we recognized that this approach might not appeal to all the undergraduates. Hence, the students were encouraged to withdraw from the module if they were not fully committed.

The overall task details, time scales and deadlines were defined at this opening stage. To make the practical side work smoothly, a fast, reliable staff–student communication channel was needed. For this we used e-mail, setting up a group address for the course. It was made clear that this would be the normal mode of communication. All the students were required to register on the campus e-mail system. Furthermore, the onus was placed on the students to read their e-mail; printed information was to be used only in unusual circumstances. We did, of course, ensure that our system worked correctly—both technically and administratively—before using it in earnest.

Task 1

As stated earlier, task 1 was carried out on an individual basis. Its objectives were both general and specific, including the following.

1. To enable the student to develop design and implementation skills through a learning-by-doing process.
2. To foster the use of a rigorous and sound approach to software design.
3. To expose the students to programming for embedded systems.
4. To get the students to produce testable software, together with appropriate test methods.
5. To introduce the concept and practicalities of design documentation.
6. To introduce, in a practical way, the idea of separating application and service (support or library) software.
7. To put formal methods in a practical context.
8. To have the students appreciate the role and use of CASE tool technology.

The students were told clearly what was expected of them in the task-defining document in Appendix 1.

Task 2

Task 2 is a group activity. Its outcome is a set of design not implementation documents (that is, no source code). Further, in deriving the design, each group must incorporate a set of the service modules developed in task 1. The group is also required to award a mark for each of these modules. This complete exercise has a number of objectives, the major ones as follows.

1. To force the students to make a clear separation between design and implementation.
2. To demonstrate how service (library) software affects application software.
3. To bring home the need for good, clear meaningful interfaces to service modules.
4. To make the students assess the source code in order to learn the problems of working with existing software.
5. To show the need for comprehensive documentation if the software is to be reused and/or maintained.
6. To demonstrate the value of CASE tool methods and documentation for a team development.
7. To highlight the need for system-level testability and test plans.

The overall task aspects are defined in Appendix 2.

The application task is to produce a design for operating the games box of Fig. 4. Fuller details of the game, together with information relating to the box itself, are given in Appendix 3.

The assessment scheme used on the task 1 modules is given in Appendix 4.

Task 3

The target of task 3 is to take a design generated during task 2 to a fully working state. This, like task 2, is a group exercise. There is also a requirement to assess the design supplied to this implementation group.

Here the objectives are to make students do the following.

1. Appreciate what it means to implement the design of others.
2. Realize the need, when carrying out software design, to produce comprehensive, correct and clear documentation.
3. Understand the usefulness of good test documentation.
4. See how to integrate service modules into application software.
5. Work together as a group to code, test and debug a design.
6. Further develop their presentation skills.

Assessment and moderation

The overall assessment and moderation process is shown below in Table 3 and Fig. 5. The individual service modules produced during task

Table 3. Assessment and moderation scheme

Work	Task 1	Task 2	Task 3
Assessed by	Student group during task 2	Student group during task 3	Staff and laboratory supervisors
Moderated by	Laboratory supervisors	Laboratory supervisors	

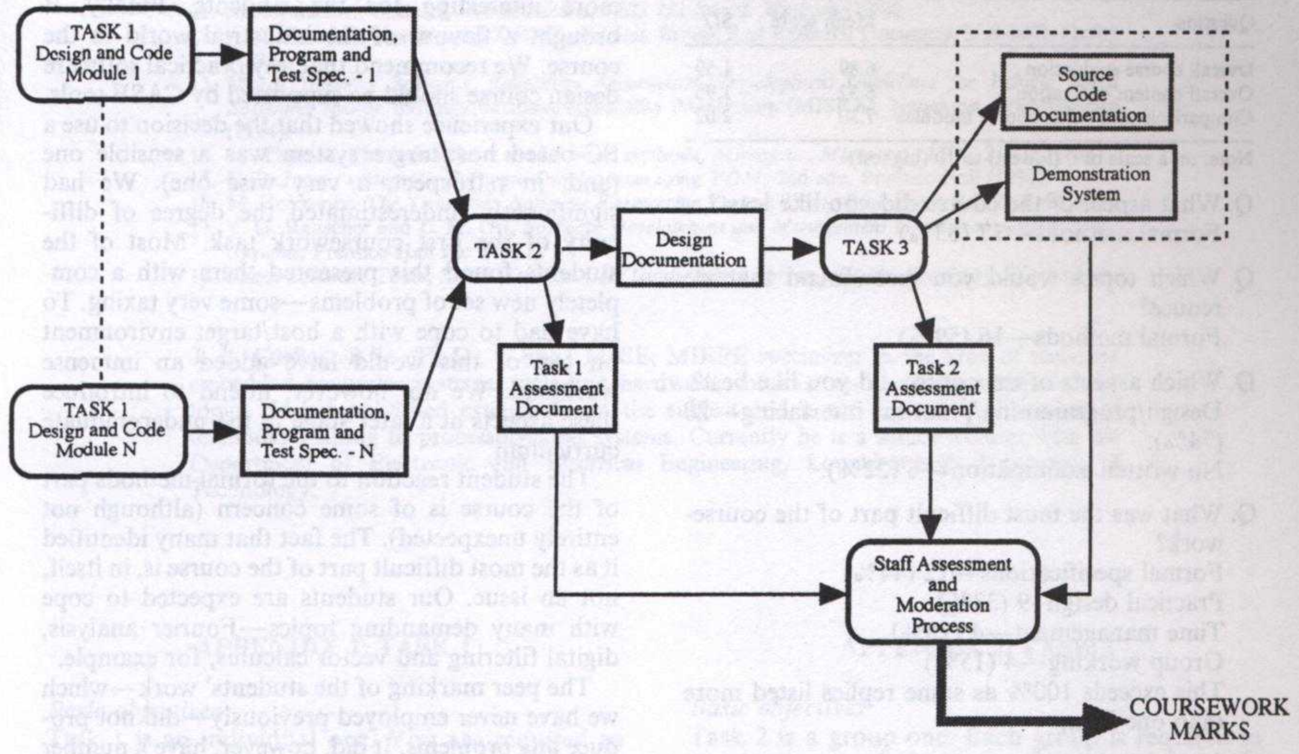

Fig. 5. Task assessment and moderation scheme.

1 are assessed by the student groups during the task 2 session. Each module is marked (quite separately) by two groups. The final mark used is the mean of the two (significant differences in marks were to be investigated, but fortunately this problem did not actually arise).

To ensure that the assessment was fair and consistent, it was essential to have a clear set of guidelines. Those for task 1 are given, in full, in Appendix 4. It can be seen that considerable thought and effort has to be put into such documents. Further checks were carried out by the supervisors using the moderation document of Appendix 5.

Similar documents were produced for the other tasks.

COMMENTS AND CONCLUSIONS

General comments

This approach to coursework produced two major surprises. First, we had no significant problems with running the module; all went fairly smoothly. Second, 10 of the 11 student groups actually produced fully working, demonstrable and correct games software (this, I might add, is a tribute to our students).

The degree of interest and enthusiasm shown by the students was consistently high (and, for us, invigorating). This was maintained in spite of a very heavy workload and tight deadlines. They found working with real-world devices to be a challenging task. It also brought a new perspective to software development, being quite different to earlier experience.

One of the most valuable aspects of the coursework was exposing the students to the work of others. It is no exaggeration to say that some students were startled by the contents of such designs. The move from producer to consumer effected a clear shift in attitudes. At the beginning, most (if not all) the students considered software design to be synonymous with program design. On completion of the course, most had changed their views. They understood software design to be a rigorous process, quite different to programming (unfortunately, there are some who will forever remain hackers).

We also, in a small way, narrowed the software–hardware divide which is so commonplace nowadays.

The student viewpoint

The module was attended by students from three undergraduate courses. For one course the module was compulsory, while for the others it was optional. To obtain feedback on the module, we sent out a questionnaire to all the participants. Many aspects were covered in the questionnaire, with most relating to the internal and organizational aspects. The following are perhaps the ones most relevant to this paper.

Question	Mean score	SD
Overall course evaluation	6.89	1.59
Overall content evaluation	6.52	1.95
Comparison with other course modules	7.30	2.02

Note: on a scale of 0 (lowest) to 10 (highest)

Q. What aspect of the course did you like least?
Formal methods—17 (63%)

Q. Which topics would you recommend that we reduce?
Formal methods—16 (59%)

Q. Which aspects of the course did you like best?
Design/programming/practical interfacing—20 (74%).
No written examination—6 (22%).

Q. What was the most difficult part of the coursework?
Formal specifications—12 (44%)
Practical design–9 (33%)
Time management—4 (15%)
Group working—4 (15%)
This exceeds 100% as some replies listed more than one item.

Q. Would you recommend this course to others?
Yes—23 (85%)
No—4 (15%)

Conclusions

One of our prime objectives was to get the students to recognize the importance of software design. Based on both formal and informal feedback, we believe this has been achieved with great success. Its acceptance by the students had much to do with applying design techniques to a real (and real-world) problem. Further, they ended up appreciating the need for—and use of—related design documentation.

The use of a CASE tool in the coursework had many advantages. First, it enforced a consistent and rigorous approach to design. Second, it took the tedium out of producing design diagrams and supporting paperwork. Third, it made the work more interesting for the students. Finally, it brought a flavour of the industrial world to the course. We recommend that any practical software design course should be supported by CASE tools.

Our experience showed that the decision to use a PC-based host/target system was a sensible one (and, in retrospect, a very wise one). We had significantly underestimated the degree of difficulty of the first coursework task. Most of the students found this presented them with a completely new set of problems—some very taxing. To have had to cope with a host/target environment on top of this would have added an immense workload. We do, however, intend to introduce these aspects at a later stage in the undergraduate curriculum.

The student reaction to the formal methods part of the course is of some concern (although not entirely unexpected). The fact that many identified it as the most difficult part of the course is, in itself, not an issue. Our students are expected to cope with many demanding topics—Fourier analysis, digital filtering and vector calculus, for example.

The peer marking of the students' work—which we have never employed previously—did not produce any problems. It did, however, have a number of benefits. In particular, both subjectively and objectively.

It can be seen from the student feedback that the course has generally been a popular one. It also shows that there is room for improvement, something we are very conscious of. In light of this, we have made a number of changes. First, the subject content of the formal methods has been slightly reduced. Second, it has been more closely allied with the practical work of the individual tasks. Third, the practical (assessed) work now starts in the first term. Task 1 is to be completed by the end of this term. Only time will tell how effective these changes have been.

Acknowledgements—Many people contributed to the development and running of this course. Three however, deserve special thanks: Paul Cherry, Mike Woodward and Alan Cuff.

REFERENCES

1. Joint Working Party of the British Computer Society and the Institution of Electrical Engineers, *IEE: A Report on Undergraduate Curricula for Software Engineering*, Joint Working Party of the British Computer Society and the Institution of Electrical Engineers (1989).
2. R. Seager, Modula-2 in a first year software engineering course, *Proceedings of the Conference on Modula-2 in Undergraduate Curricula*, Plymouth Polytechnic (1988).
3. J. E. Cooling, *Software Design for Real-Time Systems*, Chapman & Hall (1991).
4. S. Goldsmith, *A Practical Guide to Real-Time Systems Development*, Prentice-Hall (1993).
5. DEF-STAN 00–55, The procurement and use of software for safety critical applications (1989).
6. K. Shumate, Structured analysis and object-oriented design are compatible, *Ada Lett.*, **XI**(4), 78–90 (1991).
7. D. Firesmith, Structural analysis and object-oriented development are not compatible, *Ada Lett.*, **XI**(9), 56–66 (1991).
8. A. Collins, *The Object-oriented Design of Real-time Embedded Systems*, Project Report, Department of Physics, Loughborough University of Technology, Loughborough (1993).
9. F. Detienne, Difficulties in Designing with an Object-oriented Language: an Empirical Study, *IFIP Human–Computer Interaction–INTERACT'90*, pp. 971–976.
10. T. Grechenig and S. Biffl, The challenge of introducing the object-oriented paradigm—am empirical investigation of a software-engineering course, *Struct. Program.*, 14, 187–198 (1993).

11. Motorola Inc., Training and education, *ENG 232 Struct. Methods* (1989).
12. A. C. Dumbill, *Design using DFD Techniques*, British Rail Research Document SCS-1001, Derby (1993).
13. Motor Industry Software Reliability Association, *Development Guidelines for Vehicle Based Software*, Motor Industry Software Reliability Association (MISRA), Nuneaton, Warwickshire (1994).
14. M. Thomas, The industrial u se of formal methods, *Microproc. Microsyst.*, **17**(1), 31–36 (1993).
15. C. B. Jones, *systematic Software Development using VDM*, 2nd edn, Prentice-Hall (1990).
16. M. Holcombe, *The 'Crossover' Software Engineering Project, Engng Prof. Council Bull.*, 3–5 (1994).
17. T. G. Rauscher and L. M. Ott, *Software Development and Management for Microprocessor-based Systems*, Prentice-Hall Inc. (1987).
18. Select Software Tools, *Select Yourdon User Guide*, Select Software Tools, Prestbury, Cheltenham.

J. E. Cooling, B.Sc., Ph.D., C.Eng., MIEE, MIEEE specializes in the area of real-time embedded computer systems, including hardware, software and systems aspects of the topic. He has published extensively on the subject and is the author of a number of textbooks relating to processor-based systems. Currently he is a senior lecturer with the Department of Electronic and Electrical Engineering, Loughborough University of Technology.

APPENDIX 1: TASK 1

Basic objectives

Task 1 is an individual one. You are required to develop one specific software module as a 'service' module. Your work should demonstrate the following aspects of software design.

1. Structure design principles.
2. Source code (program) aspects—style, layout, naming, commenting, abstraction, information hiding, interfaces, application of pre- and post-conditions and provision of test information.
3. Modularization, with special reference to 'service' modules.
4. Design based on diagramming techniques.
5. Developing software as a set of abstract software machines using structure chart principles.
6. The Select CASE tool for developing and recording the design.

Deliverables

1. All design and code information on disk.
2. A test program to demonstrate the working of your design. This should be fully documented.

Technical details (part)

Switch in module. This module must provide facilities for the application programmer to perform the following.

1. Read in the state of all switches.
2. Read in the state of any group of switches.
3. Read in the state of any individual switch.
4. Enable and disable the switch input interrupt.

Switch signal debouncing should be done in software, but the application programmer must be able to override this feature.

APPENDIX 2: TASK 2

Basic objectives

Task 2 is a group one. Each group is required to develop application software for the game's exercise described below. To do this it will be necessary to exercise various functions of the interface board. The software is to comprise modules developed by other students for task 1 and an application (monitoring, control and display) task which uses these modules.

You are required to demonstrate an understanding and use of structured software design techniques incorporating real-time data flow diagrams, control transformations and associated state transition diagrams, process specifications and data dictionaries and structure charts (specifically, the Jackson types as defined in the Select CASE tool).

APPENDIX 3: THE APPLICATION TASK DETAILS (PART)

Overview

This document describes a reaction challenge game that compares the performance of two competitors in a simple contest of speed and dexterity. In a series of plays, each competitor must race to complete a simple task more quickly than their opponent. Winning requires both accuracy and speed, the absence of either invokes a penalty. The game uses a PC with parallel digital and analogue I/O channels, which have to be programmed to achieve the specific functionality.

The core of the system is a games box. This contains the following.

1. A stepper motor fitted with a small lightweight disc fixed so that the disc rotates in a horizontal plane. Sited around the circumference of the disc are eight slotted optocouplers mounted so

that they pass a signal only when a slot cut in the edge of the disc passes by.
2. Eight LEDs, marked '0–7' are arranged in a circle.
3. Analogue voltmeters.
4. Various switches and potentiometers.

On system start up, the PC de-energizes the digital outputs and sets the analogue output to zero. The user is invited to enter parameters via the keyboard to set up the following.

1. The number of points to be played.
2. The timeout on each point.
3. The score ratio for the analogue input (see later paragraph).

Pressing the start key on the keyboard causes the system to run the stepper motor for a defined number of steps. The number of steps should be generated by a simple pseudo-random number generator. When it stops, the slot in the disc will be aligned with one of the eight slotted optocouplers mounted around the periphery. The system reads its identity and the system lights the corresponding LED.

Each competitor is furnished with three switches, a push-button and a multiturn potentiometer. Three simple analogue meters are also mounted on the console.

As soon as one of the LEDs lights up, each competitor must race to set their three switches to the binary equivalent of the LED number, then must turn the potentiometer so that their meter matches the one driven by the system as closely as possible. When satisfied, they press their push-button once to input the values to the system. The system only accepts input on the first press of the button in any play.

On detection of the first button press from either player, the system freezes the analogue output and inputs the analogue value from that player's potentiometer.

APPENDIX 4: TASK 1 ASSESSMENT DOCUMENT

Assessment points and marks allocation (%).

Documentation supplied	10
Working software	30
Software design	30
Code implementation	20
Formal specification aspects	10

Where written comments are required (or you wish to make some), print these on a separate sheet(s)—the comment sheet. All comments should be numbered to correspond with the appropriate assessment point.

Documentation supplied

What you are assessing here is the extent, completeness and usability of the documentation. Aspects relating to the specifics of contents are covered in other sections.

1. Was the documentation well produced in terms of appearance, binding and robustness?
2. Was it complete? If not, list what was missing?
3. Was it clearly identified, including the name of the author?
4. Was it easy to use the material as a working document?
5. Was it easy to navigate your way around the document?
6. Was it organized into sensible sections (e.g. design, code, etc.)?
7. Were all the figures given identifying names and numbers?
8. Did it clearly and accurately cross-reference the software supplied on disk?
9. Did it clearly explain what the various disc files contained?
10. Did it include a table of contents, a list of figures, an overall description of the structure of the document and a description of the topics contained within each section.

Subtotal 1: Marks out of 10

Working software

1. The software worked correctly, the module was correctly built as a service module and the testing was complete, thorough and straightforward.
   Award 30 Marks.

OR

2. The software worked correctly when tested but could only be used with the test suite (i.e. it failed to meet its objective of being a service module). Testing was straightforward.
   Award 15 Marks.

OR

3. The software was only partly complete. However, that which was done could be tested out.
   Award marks out of 15.
   List your difficulties in assessing and testing the software on the comment sheet.

4. The software did not work.
   Award 0 marks.

Subtotal 2: marks out of 30

Software design

Part (a).

1. Did the text clearly and accurately explain the purpose and function of the module?
2. Did it clearly describe the structure of the module and the reason for such structuring?
3. Did it clearly explain how to use the service module and its component parts?
4. Were you able to understand the design struc-

ture charts (SCs) without the assistance of a text description?

5. Were you able to understand the essentials of the design without the assistance of a text description.
6. Did the SC design conform to the rules laid down for Jackson (or modified Jackson) structure charts? See also the last item—code skeletons.
7. Was the naming used on the SCs clear and sensible. Did these, especially at the higher levels, relate to the system details (rather than code implementation aspects)?
Marks out of 15

Part (b).

1. Code skeleton: generate the program code skeletons from the SCs provided. Extract the corresponding actual code skeleton from the submitted code (copy and then modify the appropriate disk file information). Attach both to this assessment sheet.
   (a) There is very good correspondence between the skeleton defined by the SCs and that actually produced in the source code.
   Award 15 marks.
   (b) There are important differences between the two.
   Award marks out of 15
   List the more important points on the comment sheet.
   (c) There is no correspondence between the two.
   Award 0 marks.
   Subtotal 3: marks out of 30

Code implementation

Part (a).

1. Was the software concisely housed within a library module?
2. Could you have easily integrated this as a new library module to extend that standard ones of the Stony Brook compiler?
3. Did the definition module contain a clear description of the function, features, objectives, etc., of the module.
4. Did the definition module contain only those items required for export?
5. Were items made visible in the definition module which were used **only** within the implementation module?
6. Was the definition module easy to read and assess?
7. Was a clear, comprehensive and meaningful description provided for all exported items?
8. Was the information given in the definition module sufficient to allow an implementation to use the provided service facilities? Or would you have to resort to reading the details of the implementation module?
9. Were good abstract interfaces provided? Or would you have to know considerable details of the interface board to use the service software?
10. Did the definition module include the author's name, issue date and issue version?
   (a) The service module software was packaged as a library module.
   Award marks out of 10
   (b) The service module software was not packaged as a library module.
   Award 0 marks.

Part (b).

1. Did the implementation module include the author's name, issue date and issue version?
2. Was a good, clear layout style used?
3. Was the naming logical and meaningful? Did it lead to a self-documenting code?
4. Was the level and amount of commenting satisfactory?
5. Did the names used within the program relate (where applicable) to the corresponding SC items?
6. Was it easy to follow and understand the code? Would it have been easy to perform a code walkthrough exercise?
Award marks out of 10
Subtotal 4: marks out of 20

Formal specification aspects

Part (a): VDM specification.

1. Does the specification describe a loop?
2. Are the pre- and post-conditions given in the specifications?
3. Are the pre- and post-conditions valid logical algebraic expressions?
4. Is the specification adequately explained by a suitable comment?

Award 1 mark for Yes answers to each of 1–3. Award 2 marks to a yes answer to 4. Each no answer gets 0 marks.

Marks awarded

Part (b): correctness proof.

1. Have the pre- and post-conditions been carried through from the VDM specification?
2. Has the proof been correctly decomposed into its constituent parts
3. Is the proof correct?

Award 1 mark for yes answers to each of 5 and 6. Award 3 marks to a yes answer to 7. Each no answer gets 0 marks.

Subtotal 5: marks out of 10

Bonus aspects
Answer the questions only, no marking required.

1. Was a context diagram produced?
2. Was it accurate, meaningful and useful?
3. Was a data flow diagram produced?
4. Did it clearly express the functionality of the service module?
5. Did you find it useful?
6. Was a state transition diagram produced?
7. Was it accurate, meaningful and useful?
8. Were you able to see how the SC was generated using the information given in the DFDs and the STDs?
9. Was the relationship between the diagrams obvious?

APPENDIX 5: TASK 1 MODERATION DOCUMENT

1. Are all the sections of the document complete?
2. Has the fifth section (formal specification) been marked (if not, note this in red)?
3. Has the sixth section been dealt with adequately?
4. Do the marks total up correctly?
   If not, note in red. Identify, if possible, where the problem is.
5. Is there any sign of personal bias?
6. What is the mark for this module?
7. What mark was given for the other copy?

If there is a difference of 20 marks or more between these, note this in red. Then investigate the problem (allow for the marking of formal specifications). Attach your comments to this document.

8. What is the average of the two marks?
9. How well was this module marked (marks out of 100)?