ISSN 0254-2757

QI QUÆSTIONES INFORMATICÆ

Volume 5 • Number 1 April 1987

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereeneging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUÆSTIONES INFORMATICÆ

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Subscriptions

Annual subscription are as follows:

	SA	US	UK
Individuals	R10	$ 7	£ 5
Institutions	R15	$14	£10

Quæstiones Informaticæ is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

EXPERIENCE WITH TEACHING SOFTWARE ENGINEERING

C H Hoogendoorn
Department of Computer Science
University of the Witwatersrand
Johannesburg

1. INTRODUCTION

The term "software engineering" has been in use for nearly 20 years since the so-called "software crisis" was identified in the late 1960's. However, it is only comparatively recently that courses in software engineering have begun to be taught in university computer science departments. Various reasons have been cited for the slow acceptance of software engineering as a subject in a computer science curriculum, for example the lack of suitable textbooks, the lack of suitably qualified staff, a perceived conflict between the "scientific" and "engineering" approaches to computing as well as students' lack of experience with large scale software and their resulting inability to appreciate the problems.

Fortunately, the situation with respect to textbooks has greatly improved. Moreover, a number of articles have appeared (Kant [1], Sommerville [2], Collofello and Woodfield [3], Lee and Frankel [4], Wu [5], Petricig and Freeman [6]) reporting on software engineering courses introduced at universities at the graduate and undergraduate levels. The general view which emerges is that such a course should have a very sizeable practical component in the form of group projects in an attempt to bridge the gap between one-person programs completed in the space of weeks and larger-scale software development undertaken by teams. Group size and project duration vary from 5 persons for one semester to 7-10 persons over 4 semesters. The scope of the projects varied from covering the entire life cycle (incuding maintenance), analysis and design only, development cycle only (excluding maintenance) or design and implementation from given requirements.

Encouraged by these reports, the Department of Computer Science at the University of the Witwatersrand decided to introduce software engineering as a topic in computer science in the final (3rd) year of undergraduate study. At this stage students will have had two years of programming experience in Pascal as well as some assembly language programming and will concurrently be taking topics in programming languages, translators, data bases and theory of computation. The objectives which we have set ourselves are fairly modest. The aims are to demonstrate the difficulties of software specification, design, development and testing, to demonstrate the problems of working in a team and to develop written and oral communication skills. In common with most authors we do not believe that a university can produce fully-trained software engineers. However, students should at least become acquainted with the problems of large-scale software production and become aware of solution approaches. An important side benefit of such exposure is that it leads to a better appreciation of developments in various areas of computer science in answer to the software crisis. Examples are modularity and abstract data types in programming languages, protection in operating systems as well as program verification.

2. COURSE ORGANISATION

The software engineering course extends over the entire academic year, starting in late February and terminating in late October. During this period the students, working in project groups, are expected to take a software system through the entire development cycle, i.e. requirements analysis, design, implementation (coding), testing and integration as well as delivery and demonstration of the final product. The project really serves as a means to an end: while workingon the project students will be confronted with the problems of developing larger software systems and will be exposed to software engineering principles, methods and techniques for dealing with these problems.

As mentioned in the introduction, students will be taking other topics in Computer Science apart from software engineering. Moreover, a student will typically be registered for another third-year course as well as a first year course. A realistic time budget per student for the course, taking into account examinations, mid-term breaks etc., is between 160 and 200 hours per student. Assuming 5 students per project group, this leads to a typical project size of between 5

Quæstiones Informaticæ, Vol 5 No 1, PP 36-40, ISSN 0254-2757

and 6 man-months. While small by "real-world" standards, it is still large enough for scaling up effects to become visible and certainly much too large to be completed by a single individual. The scope of the projects, i.e. the entire development cycle, was chosen in order that the consequences of decisions taken in the analysis and design phases could be experienced during implementation, testing and integration. However, as Sommerville [2] points out, a disadvantage of this choice is that the size of the project must be smaller (and therefore perhaps less representative of real-world problems) compared to an approach which only includes analysis and design.

Students are offered a choice of projects to work on. Typical projects that have been offered include the following:

(i) An office diary and calendar system, including facilities for the scheduling of group meetings.
(ii) A lecture room reservation system for a University.
(iii) An interactive text formatting system.
(iv) A simple computer-aided instruction system for teaching nurses to reason about diseases and symptoms.

A common feature of most projects is that they call for the development of a substantial man-machine interface, thus exposing students to this most critical design area. It is endeavoured to choose project topics which minimize the amount of "outside" (non-sofware-engineering) knowledge to be acquired. All projects are open-ended to some degree; this avoids the difficulty of choosing a project of exactly the right size.

At the beginning of the year students are asked to rank the projects in order of preference. They are then allocated to groups according to the following criteria:

(i) Project preference.
(ii) Commonality of free timetable slots (essential for group work!)
(iii) Performance in previous computer sience courses. A mix of high and average achievers is aimed for in each group.

It is expected that all group members participate in all aspects of development, i.e. analysis, design, coding, testing, documentation etc. Although no fixed group structure is imposed, groups are very strongly urged to choose a leader who can be responsible for allocating tasks and monitoring their completion, making sure that deadlines are met and so forth. Group leadership can either remain fixed or can be rotated between group members. However, in spite of advice to the contrary, some groups choose to function as a democratic team without a leader and almost invariably report later that this was a big mistake! Very similar experiences have been reported by Kant [1]. Groups are encouraged to keep minutes of meetings held amongst themselves, recording decisions which were made, problems which were identified, the allocation of specific tasks etc. Each student is also encouraged to keep a log book in which all time spent on project activities is recorded, along with the type of activity. These log books are intended to provide students with more insight into the development process and are not intended to be scrutinized by staff. No incentive for falsification therefore exists.

For each project a staff member is appointed as supervisor, which implies acting both as a customer and a consultant. Regular meetings are scheduled with the supervisor throughout the year. In addition the lecturer in charge of the course (the projects coordinator) also has regular meetings with each group throughout the year, usually addressing general software engineering issues rather than project-specific ones.

No formal lectures are given. A book on software engineering (Sommerville [7]) is prescribed, which students are expected to read in their own time. In addition, a series of project notes are issued by the projects coordinator throughout the year. These notes contain inter alia some practical hints for the applicable phases of the development cycle, as well as suggestions for further reading. This material is often used as a basis for discussion in the group meetings with the projects coordinator.

Projects are implemented on a wide variety of equipment, ranging from the university mainframe to departmental minicomputers and personal computers. The choice of equipment is usually tied up with the project choice. Students are urged to familiarize themselves at the earliest opportunity with the editors, language translators, data base systems, word processors and other software tools available on each type of equipment. It is required that all documentation be done on-line. Fortunately, the introduction of the course coincided with the acquisition of some very

good facilities for word processing, text formatting and printing (laser), so that the insistence on on-line documentation by now seems superfluous.

Although students are made aware of various methodologies for analysis, design and so forth, we do not insist on the use of any particular methodology. This is in line with our objective of exposing students to software engineering problems before introducing specific solutions. In any event, such an insistence would be hard to adhere to in practice, given the fairly diverse nature of projects and equipment, as well as our lack of support tools for specific methodologies.

A number of project milestones are set throughout the year, corresponding with the various phases of the development cycle, i.e. analysis, design, coding, test and integration and delivery and demonstration of the final product. The design stage is further subdivided into overall and detailed design. A deliverable item (e.g. a specification) is due for each project milestone. Two copies of all documents are required, one of which is marked by both the project supervisor and projects coordinator. This copy is returned to the group and discussed with them as soon as possible, in order to provide the necessary feedback for the next phase.

The most intensive interaction between groups and their supervisors and the projects coordinator occurs in the requirements analysis phase. During this phase the supervisor acts mostly as customer, while at the same time ensuring that groups not only address the most obvious functional requirements but also pay attention to non-functional requirements such as reliability and performance. Much emphasis is placed on the overall organization of the SRD (Software Requirements Document), with good use of numbering schemes and indexing to allow easy future identification of specific requirements. It seems good practice to let one group member assume overall responsibility for the SRD (and its timely delivery), in order to ensure a uniform notation. Review meetings are encouraged in order to achieve consistency and completeness.

A similar emphasis is placed on the Overall and Detailed Design Documents. The preferred notation for the former is some form of structure diagrams augmented by structured descriptive paragraphs, while for the detailed design pseudo-code is normally employed. Considerable guidance is usually necessary for system decomposition and the definition of interfaces between subsystems, while the need for supervisor and coordinator interaction decreases once detailed design and coding commences. At this stage the regular meetings are most useful to monitor progress and to identify technical and non-technical problems as early as possible.

Shortly after the detailed design has been completed, a Test and Integration Document is due. The intention here is to force students to think about testing strategies and methods. The use of code reviews and design walkthroughs is highly encouraged. As part of this document an integration plan is required, which in turn will determine the order in which modules are developed and tested. Before the final delivery milestone an informal interim milestone is usually defined. At this stage a working system which demonstrates at least the main system functions is expected.

For final delivery, a complete copy of the software system on some suitable medium (tape, floppy disk) as well as complete documentation, including an installation guide, user manual, operator/system administrators guide (where relevant) is expected. Shortly afterwards, live project presentations are held, using screen projection equipment, in front of the entire class, departmental staff and selected guests.

All group deliverables and presentations contribute 50% towards the mark for each student. Half of this in turn, i.e. 25% of the total, is accounted for by the delivered final product and its presentation, the bulk of the remainder being accounted for by the SRD and Design Documents. A further 25% of each student's mark is for individual contribution to the group effort. This assessment is based on observations by the supervisor and projects coordinator, as well as a statement of contributions provided by each student. The remaining 25% is for an individual essay on software engineering which is due two weeks after the final product deadline. The structure and format of this essay is entirely open-ended, the idea being that it should reflect what the student learnt about software engineering in the course of doing the project, including managerial issues, problems of working in a group etc. Comments on the course and suggestions for the future are also invited.

3. EXPERIENCE WITH THE COURSE

Generally the feedback which is received indicates that students find the course very worthwhile, some going as far as claiming that they learnt more from it than any other course! Initially a few students expressed some misgivings about being assessed on a group basis rather

than individually. However, the even balance between group and individual assessment, as well as the generally high marks which are obtained have served to allay these fears. The high marks in turn reflect the generally high standard achieved in the final product, documentation and demonstration. Many students seem prepared to put in large amounts of work, in some cases much more than could reasonably be expected (and, one suspects, more than they would have done if they had been working as individuals).

Most students considered the requirements analysis phase, leading up to the production of the SRD, one of the most valuable project experiences. Groups which produced a high-quality SRD generally saw this as a key to their success on the entire project. On the other hand, groups which did less well here often found that it required much time-consuming effort to recover from the effects of a bad SRD. However, even in those cases where a high-quality SRD was produced, it was often found later in the development cycle that requirements needed to be modified due to over-optimistic time estimates, unanticipated problems with systems software, insufficient understanding of the system to be developed, machine performance problems and similar causes. While some of these pitfalls would have been avoided by more experienced analysts, others could only have been detected through prior experimentation. For this reason, as of this year the project development cycle has been modified to include prototyping. The project is now kicked off with a small feasibility study, after which a prototype, demonstrating some aspects of system functionality and man-machine interfacing, is developed. The SRD, which remains the first formal document to be submitted, is now due a few weeks after prototype demonstration. As the total time for the project remains inelastic, this change implies that less time is available for design, implementation and testing. However, it is felt that the experience gained in prototyping will compensate for this loss.

The overall design phase was also considered to be a very valuable learning experience. Many groups found the partitioning of a system into subsystems with well-defined interfaces a non-trivial task, requiring perhaps a number of iterations as well as guidance from staff members. On the other hand, the detailed design seemed to be far less problematic (and less valuable), often descending to a level not far removed from executable code. As it did seem to be somewhat of an overkill for the given size of project, it was therefore decided to do away with detailed design as a separate milestone. Somewhat more detail is now required from the Overall Design (now simply Design) Document, after which implementation commences.

While some groups seemed to derive real benefit from producing a Test and Integration Document, others seemed to regard it as little more than a nuisance; the actual development order and test methods bearing little resemblance to those described in the document. At this stage students became very conscious of the pressure to complete the work by the final deadline. As a result, testing and integration became rather ad hoc, sometimes with disastrous consequences, requiring extreme effort to recover. Although much advocated, code reviews became the exception rather than the rule. It is therefore planned that the project supervisors and coordinator shall play a more active role here, for example by conducting code reviews with a group.

With the increasing diversity and complexity of university computer facilities, it was observed that some groups required a long time to become familiar with the hardware and software available for project implementation (including documentation). This in turn led to decisions being taken during the analysis and design stages which subsequently had to be overturned (with much loss of effort) as more experience and information became available. The change to include prototyping, thus getting hands dirty at an early stage, is expected to alleviate this problem. It was also decided, in the course of restructuring the undergraduate Computer Science syllabus, to offer a series of lectures on "Software Tools" in the first semester of the 3rd year of study. These lectures are given by staff members as well as experts from elsewhere (e.g. Computer Centre, industry) and cover a wide variety of topics, for example:

1. Specific programming languages (Fortran, APL, Cobol, LISP etc)
2. Specific documentation facilities (e.g. DCF)
3. Project management
4. Specific database systems and packages (e.g. SAS)
5. Writing skills.

None of this material is formally examined, although assignments may be set in some areas, e.g. programming exercises in specific languages. In addition to providing useful input for the software engineering project, the material is a prerequisite for some other courses. For example, the Programming Languages course (offered in the second semester) can now concentrate on language principles in the knowledge that students will have had prior exposure to a number of

than individually. However, the even balance between group and individual assessment, as well as the generally high marks which are obtained have served to allay these fears. The high marks in turn reflect the generally high standard achieved in the final product, documentation and demonstration. Many students seem prepared to put in large amounts of work, in some cases much more than could reasonably be expected (and, one suspects, more than they would have done if they had been working as individuals).

Most students considered the requirements analysis phase, leading up to the production of the SRD, one of the most valuable project experiences. Groups which produced a high-quality SRD generally saw this as a key to their success on the entire project. On the other hand, groups which did less well here often found that it required much time-consuming effort to recover from the effects of a bad SRD. However, even in those cases where a high-quality SRD was produced, it was often found later in the development cycle that requirements needed to be modified due to over-optimistic time estimates, unanticipated problems with systems software, insufficient understanding of the system to be developed, machine performance problems and similar causes. While some of these pitfalls would have been avoided by more experienced analysts, others could only have been detected through prior experimentation. For this reason, as of this year the project development cycle has been modified to include prototyping. The project is now kicked off with a small feasibility study, after which a prototype, demonstrating some aspects of system functionality and man-machine interfacing, is developed. The SRD, which remains the first formal document to be submitted, is now due a few weeks after prototype demonstration. As the total time for the project remains inelastic, this change implies that less time is available for design, implementation and testing. However, it is felt that the experience gained in prototyping will compensate for this loss.

The overall design phase was also considered to be a very valuable learning experience. Many groups found the partitioning of a system into subsystems with well-defined interfaces a non-trivial task, requiring perhaps a number of iterations as well as guidance from staff members. On the other hand, the detailed design seemed to be far less problematic (and less valuable), often descending to a level not far removed from executable code. As it did seem to be somewhat of an overkill for the given size of project, it was therefore decided to do away with detailed design as a separate milestone. Somewhat more detail is now required from the Overall Design (now simply Design) Document, after which implementation commences.

While some groups seemed to derive real benefit from producing a Test and Integration Document, others seemed to regard it as little more than a nuisance; the actual development order and test methods bearing little resemblance to those described in the document. At this stage students became very conscious of the pressure to complete the work by the final deadline. As a result, testing and integration became rather ad hoc, sometimes with disastrous consequences, requiring extreme effort to recover. Although much advocated, code reviews became the exception rather than the rule. It is therefore planned that the project supervisors and coordinator shall play a more active role here, for example by conducting code reviews with a group.

With the increasing diversity and complexity of university computer facilities, it was observed that some groups required a long time to become familiar with the hardware and software available for project implementation (including documentation). This in turn led to decisions being taken during the analysis and design stages which subsequently had to be overturned (with much loss of effort) as more experience and information became available. The change to include prototyping, thus getting hands dirty at an early stage, is expected to alleviate this problem. It was also decided, in the course of restructuring the undergraduate Computer Science syllabus, to offer a series of lectures on "Software Tools" in the first semester of the 3rd year of study. These lectures are given by staff members as well as experts from elsewhere (e.g. Computer Centre, industry) and cover a wide variety of topics, for example:

1. Specific programming languages (Fortran, APL, Cobol, LISP etc)
2. Specific documentation facilities (e.g. DCF)
3. Project management
4. Specific database systems and packages (e.g. SAS)
5. Writing skills.

None of this material is formally examined, although assignments may be set in some areas, e.g. programming exercises in specific languages. In addition to providing useful input for the software engineering project, the material is a prerequisite for some other courses. For example, the Programming Languages course (offered in the second semester) can now concentrate on language principles in the knowledge that students will have had prior exposure to a number of

languages.

Although students have no say in the choice of group members, it is very rare to experience problems in this area. If strong personality conflicts exist, a reallocation can be made before groups get down to serious work. Although it is to be expected that not all group members contribute equally, intervention may be necessary if serious distortions occur. Such problems are most easily detected in the weekly meetings with the project supervisor.

In the allocation of groups, it appeared that project preference was far less important than timetable commonality and a good mixture of high and average achievers in determining group performance. In fact, some of the best results were achieved by groups working on projects which were nobody's first choice. For that reason students are no longer given a project choice but are allocated in groups which have as much free time in common as possible. The value of this becomes particularly apparent as each deadline approaches!

4. CONCLUSIONS

The general feedback obtained from students' essays indicate that the course objectives of demonstrating the problems of large-scale software development are being achieved. It appears to have made the students far more appreciative of both the need for sound methodologies for analysis, design, testing and integration, as well as developments in the general field of computer science aimed at making software development more productive and less error-prone. It is perhaps most appropriate to let a student have the final word:

> *"You mean that's ALL it does ?!"*
>
> *The biggest surprise is just how little work appears to have been done after a mammmoth amount has been done. There is no way the final product can reflect the array of emotions we have all encountered: joy, relief, anguish, helplessness, frustration, fear; the sleepless nights; the comradeship; the disagreements.*
>
> *The experience gained on working on a system that someone is actually going to use, has required a new approach to the whole concept of programming. The old techniques of "hacking" and spontaneous programming may not have disappeared yet, but they are certainly a lot less evident than they ever were before. As a look at the real world for the first time, the course has been a revelation.*
>
> *There is a distinct difference between a computer scientist and a software engineer that I never knew existed. And somewhere in it all is an art - where there is no apparent justification for doing a thing one way, it just "feels right".*
>
> *Although I was far from happy with my own personal performance, because I have made a lot of mistakes in the past few months, I feel that I have learnt from these mistakes. Hopefully I will not make the same mistakes again (I suppose I will make others). I can (almost) guarantee far better performance next time round.*

REFERENCES

1. Kant, E.,[1981] A Semester Course in Software Engineering, *ACM SIGSOFT SoftwareEngineering Notes*, **6**, 4 , pp 52-76.
2 Sommerville, I., [1983], Software Engineering - An Educational Challenge, *IFIP 83*, pp 193-197, Amsterdam: North-Holland.
3 Collofello, J.S. and Woodfield, S.N., [1982] A Project-Unified Software Engineering Course Sequence, *ACM SIGCSE Bulletin*, **14**, 1, pp 13-19.
4 Lee, K.Y. and Frankel, E.C., [1983], Real-Life Software Projects as Software Engineering Laboratory Exercises, *ACM SIGSOFT Software Engineering Notes*, **8**, 3, pp 39-43.
5 Wu, B.F., [1983], A Practical Training in Software Engineering Methodology, *ACM SIGCSE Bulletin*, **15**, 3, pp 2-9.
6 Petricig, M. and Freeman, P.,[1984], Software Engineering Education - A Survey, *ACM SIGCSE Bulletin*, **16**, 4, pp 18-22.
7. Sommerville, I.,[1985], *Software Engineering* (2nd Ed.), Addison Wesley, Wokingham, England.

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review artilces and exploratory articles of general interest to readers of the journal. The preferred languages of the journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:

Prof. G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. Manuscripts produced using the Apple Macintosh will be welcomed. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil and the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Drawings etc., should be submitted and should include all relevant details. Photographs as illustrations should be avoided if possible. If this cannot be avoided, glossy bromide prints are required.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

1. Ashcroft E. and Manna Z., The Translation of 'GOTO' Programs to 'WHILE' programs., *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255, 1972.
2. Bohm C. and Jacopini G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules., *Comm. ACM,* 9, 366-371, 1966.
3. Ginsburg S., Mathematical Theory of Context-free Languages, McGraw Hill, NewYork, 1966.

Proofs

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Only orginal papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.