INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)

SOFTWARE ENGINEERING – AN EDUCATIONAL CHALLENGE

Ian SOMMERVILLE
Department of Computer Science, University of Strathclyde
Glasgow, UK

This paper discusses the need for education in software engineering and examines the relationship between software engineering and computer science education. It looks at the problems of including courses in software engineering in a traditional computer science undergraduate curriculum. The main part of the paper describes how these problems have been tackled in the computer science department at the University of Strathclyde where courses in practical software engineering are a compulsory part of the curriculum. The paper concludes that it is impractical to teach students to be software engineers but that courses in software engineering are useful so that students are introduced to the problems which they might encounter in building large software systems.

1. INTRODUCTION

The term 'software engineering' was first brought to prominence around the end of the 1960s when it was realised that the development of large software systems was a problem which was more akin to engineering problems than to problems in mathematics or natural science. Since then, there has been a great deal of discussion about the problems of implementing large computer systems and a number of important techniques of large scale software development have been developed. These include formal or semi-formal languages for specifying software requirements, {1} {2}, software design representations, {3} {4}, and structured programming, {5} {6}.

In spite of these advances, the task of building large software systems remains immensely difficult and it is still common for large systems to be delivered late, to cost more than was originally estimated, to be unreliable, and to be inadequately documented. We do not yet completely understand many aspects of implementing these large software systems but some of the failures of today's software projects are due to the fact that existing techniques are not used in developing these systems. Those responsible for the implementation of these systems are inadequately trained in software engineering. Indeed, the Alvey committee in the UK, {7}, set up to review developments in information technology, recently identified software engineering to be an area of particular importance to the future prosperity of the nation and suggested that expenditure in software engineering education should be significantly increased.

Much of this training is best given after students have some practical experience in developing large software systems but we believe that there is also a need for software engineering education as part of university courses in computer science. These courses should demonstrate to students that software development cannot simply be equated with computer programming but that it embraces the entire software life cycle from the initial conception of the software to the maintenance and enhancement of a delivered software system. Furthermore, they should show that software production is subject to economic constraints in the same way as any other business or governmental activity.

In fact, software engineering education has been the subject of some discussion by authors such as Freeman, {8}, Fairley, {9}, and Mills, {10}. Although the views of different authors differ in detail, the consensus seems to be that software engineering education should be based on a firm theoretical foundation but should also include practical topics such as management science, problem solving, and communication skills.

In the remainder of this paper, the problems of integrating undergraduate courses in software engineering with computer science courses are discussed and this is followed by a description of the structure of the final year course in software engineering given to computer science undergraduates at the University of Strathclyde. This course is practically oriented and particular attention is paid to coursework completed by groups of students. The success of this course is assessed and the final part of the paper discusses, generally, what can be achieved by software engineering courses at an undergraduate level.

2. SOFTWARE ENGINEERING AND COMPUTER SCIENCE

Computer science and software engineering have an uneasy relationship with each other. Although they accept that a knowledge of some aspects of computer science is essential, many working software engineers are very dubious of the value of computer science courses as the topics covered there often seem to be divorced from the 'real world' of software development. Many teachers of computer science are wary of software engineering because it lacks a coherent core of theoretical principles and prefer to teach apparently obscure topics such as lambda calculus, say, because its academic rigour is clearly discernable. This attitude

comes out clearly in the ACM's Curriculum 78 proposals, {11}, where there is no explicit mention of software engineering as a distinct subject area.

Part of the problem arises because there is no generally agreed definition of either software engineering or computer science. In some institutions, computer science is treated as a very theoretical subject, in some its bias is towards computer hardware and in yet others the computer science course concentrates on the development of systems software. Software engineering, similarly, has a diversity of interpretations. In some cases, it is concerned with the hardware/software interface, in others it is the theory (such as there is) of software development, and sometimes software engineering is concerned with the practical problems of developing large software systems.

We favour the latter interpretation of software engineering and, as such, there are three problems which the developer of courses in this subject must tackle. These are:

(i) Computer science students in the early years of their course do not have the maturity or experience to realise that large software systems are not simply scaled-up versions of the small, fairly trivial programs which they write as educational exercises. It is only towards the end of their course, after they have experience in writing non-trivial programs and after using practical software systems, that they understand that software development is not simply the same as computer programming.

(ii) Many present-day computer science faculty members have a background in mathematics or natural science and do not appreciate the need for an engineering type of course. Opposition is particularly apparent when the introduction of software engineering courses means the reduction in courses in some other more conventionally accepted computer science topic.

(iii) Because developing large software systems takes a long time, it is quite impossible to simulate the process realistically in the time available to undergraduate students. This fact alone has meant that many courses have concentrated on particular aspects of software development without considering the topic as a whole.

Apart from these inherent problems, the introduction of courses in software engineering has been hindered by the lack of good undergraduate textbooks. Whilst there is a wealth of information on the subject, this tends to be gathered together in journals or as collections of papers. These papers are of great value but they do not present an integrated, coherent, and consistent view of software engineering.

Given that the need for courses in software engineering is accepted, the problem then remains of integrating these courses with the remainder of the computer science curriculum. The immaturity of students necessarily restricts these courses to the latter years of a computer science course although we believe that emphasis should be placed from the start on the production of reliable and maintainable programs. Furthermore, we do not believe that software engineering courses should themselves have a significant theoretical element but that the theory should be covered elsewhere in the computer science course. The job of the teacher of software engineering should be to show how theory can be applied and is actually a useful tool for the software engineer.

The approach which we have adopted is to provide a general introductory course in software engineering in the penultimate year of our course and to follow this with a more advanced course in their final year. These courses are compulsory as is a complementary final-year course in computer science theory. Ideally, there is probably a case for including an additional course covering management science and communication skills but finding time for such a course is often very difficult indeed. To some extent these topics are covered in systems analysis courses but this does not completely meet our needs. Not only is the orientation towards general business management rather than software management but some of our students do not take such courses as they are specialising in a more hardware oriented course.

3. SOFTWARE ENGINEERING AT STRATHCLYDE

As described above, computer science students at Strathclyde University must take two courses in software engineering - an introductory course followed by a more advanced course in their final year. The introductory course is intended to show that the development of a large software system has a number of distinct stages and it briefly discusses each aspect of the software life cycle. It also brings out the idea that reliability and maintainability are the most important characteristics of well-engineered software. The practical work associated with this course is oriented towards producing readable and reliable programs.

Our final year course tends to concentrate on the non-programming aspects of software development namely, requirements specification, software design, documentation, software economics, and software management. By this stage, most students have an adequate understanding of programming and are usually fairly disciplined in their programming style so we consider it unnecessary to discuss programming techniques here.

The course is based around a comprehensive set of lecture notes (now a textbook , {12}) whose unifying theme is the software life cycle. These lecture notes cover the following topics:

(i) Requirements specification

(ii) Software design

(iii) Programming

(iv) Validation

(v) Maintenance and documentation

(vi) User interface design

(vii) Software management

The programming part of the notes is included for completeness but most of that material has been covered in the elementary software engineering course taken in the previous year. The topics are covered in the order above with emphasis placed on requirements specification, software design, documentation, and software management. Each of these topics is allocated about one fifth of the total time devoted to the course.

The students are expected to work at their own pace through the course material. To assist them, about 100 tutorial problems have been provided ranging from very simple tasks to fairly complex design exercises. A representative selection of these tutorial problems is included in Appendix I. The material in the notes is not reiterated in lectures - there are no formal lectures associated with the course - but tutorial periods are available where the student may seek advice and guidance on the topics covered in the course notes.

All of the formal class contact associated with this course is devoted to discussion of practical work, the development of communication skills (both oral and written) and general discussion of topics in software engineering. In this respect, the course differs from most classes offered at UK universities where a major part of the time is spent in formal presentation of the coursework in lectures.

Our experience with this method of presentation is that students prefer it because they have good, comprehensive notes which can be read in advance and they can spend class periods in understanding rather than assimilating material. Teachers prefer it because it is much more stimulating (although harder work) than standing up and talking for an hour to an often unreceptive audience.

3.1 Practical work

As our approach to software engineering is a practical one, our software engineering course places great emphasis on project work. This project work has a number of objectives. Its primary objective is to show students some of the problems involved in developing large software systems particularly those which arise in specifying and costing the system. Its secondary objectives are to develop students communication skills, to reinforce material in the course notes, and to show them what it is like to work as a member of a group rather than as an individual programmer.

We believe this practical work has two fairly original features:

(i) Problems of software maintenance are demonstrated by requiring that students should modify a program which they have written in a previous year of the computer science course.

(ii) The instructor acts as a 'typical' software customer and makes conflicting and self-contradictory demands on the students. It is up to the students to recognise and reconcile these contradictions.

As we have already suggested, however, designing suitable practical exercises is very difficult indeed because of the limited time which the students can devote to the coursework. In essence, there are two alternative possibilities:

(i) The project chosen can be relatively small and well-defined so that a group of students can complete the specification, design, and implementation in the time allocated for the course.

(ii) The project can be larger and more comprehensive with only parts of the system completed by the students.

Neither of these alternatives is entirely satisfactory. If a small well-defined system is chosen, the tasks of specification and high-level design are trivial and appear to be unimportant. However, in real software systems it is these aspects of the development process which are often the most difficult and costly and we wish to demonstrate this to students. Our approach is to use a larger, more realistic example but this has the disadvantage that the students cannot complete the whole process of development because of time limitations. In fact, we expect them to concentrate on the specification, design and costing of the project.

However, before students start on this project we set them a small piece of practical work at the very beginning of the course. We ask them to modify a program which they worked on in the previous year. The idea underlying this work is to demonstrate the problems of program maintenance - for most students it is the first time that they have ever had to change an existing program which they had not seen for some time. Their reactions have indicated that this is a very useful exercise demonstrating the value of disciplined programming and sensible program commenting. It emphasises that maintenance can be a time consuming and difficult activity.

The main project work centres round an example of a fairly large but comprehensible system. Examples which have been used include an electronic mail/teleconferencing system and a real-time patient monitoring system such as might be used in a hospital intensive care unit. An example of a typical specification is given in Appendix II.

The initial information given to the students is intentionally vague as it is intended that each project group must question the instructor to ascertain the detailed requirements of the system. The instructor attempts to act like a typical user during this questioning phase deliberately suggesting impossible objectives

and contradictory requirements. The students must identify these in their report as well as providing a full specification of what the system is actually required to do.

The work is carried out in project groups of three or four students and each group must submit three reports:

(i) The detailed requirements of the system must be defined using a semi-formal requirements definition language.

(ii) The high-level design of the system must be specified. This document sets out the distinct parts of the system and their interrelationships but does not describe detailed algorithms which might be used. The design notation used is similar to that described by Constantine and Yourdon, {4}.

(iii) Finally, each group must complete a report setting out the system costs, and the time which would be required to complete the system. As part of this report, they must prepare a PERT chart showing the dependencies of the different parts of the system and the time required to complete each part.

As one of the objectives of the course is to develop students' communication skills, particular emphasis is placed on the quality of the technical description in all of these reports. The instructor takes care to provide detailed comments on technical writing style and report layout. We have noticed that there is a marked improvement in the quality of technical writing as students assimilate these comments in initial reports and bear them in mind when preparing later reports. In order to develop oral communication skills, all students are required at some time in the course to make a presentation in front of the class describing some of the work which they have done.

There are, of course, a number of disadvantages associated with this approach to practical work:

(i) It requires a great deal of work on the part of the instructor to mark and to comment in detail on project reports. This marking must be completed quickly so that the comments are useful for the student in subsequent reports.

(ii) There can be problems with the assessment of group projects where one member of a group does not play a full part in the work. Other members feel resentful if that individual is assessed on the same level as they are. Although this is obviously a potential problem, our experience is that it is relatively rare.

(iii) This type of project organisation is probably only viable with a relatively small class size. For large classes, it is unrealistic to expect all students to make presentations about their work simply because of the time required for such talks.

In spite of these disadvantages, we believe that the approach which we have adopted to software engineering practical work meets our primary objective namely to demonstrate the difficulties of software specification, design, and costing. It also meets our secondary objectives of reinforcing the course material , developing communication skills, and showing students some of the difficulties of group working. This supposition is based on comments by students about the course, their performance in the practical work where a marked improvement is shown from the first project to the last, and their performance in formal examinations where a fairly high class average mark has been attained.

As far as the students are concerned, informal surveys have shown that they find the project work challenging and an interesting change from programming. Their performance in the practical work is significantly better than in examinations and, as practical work makes up 1/3 of the overall course assessment, they improve their final mark by their practical performance. They also find that potential employers are very interested in this work and it seems to enhance their chances of finding a job on leaving the university.

4. CONCLUSIONS

We believe that undergraduate courses in computer science ought to include an element of software engineering education but that this is only worthwhile if it is included in the later years of their course. Unlike Jensen et al., {13}, we do not believe that there is much point in full-scale courses in software engineering because students do not have the maturity or experience to appreciate software engineering problems until they have considerable programming experience.

However, the instructor of software engineering courses must be careful not to delude himself that he is teaching students to be software engineers. We believe that this is an impossible objective and that practical experience in large scale software development is necessary before an individual can be considered to be a software engineer. The aim of university courses in this topic must be to prepare students for this practical experience by showing the problems which they might encounter in the development of large systems.

APPENDIX I - Tutorial Examples

The tutorial examples below are representative of the examples given to students to help them read and understand the course material.

(i) Discuss the value of simulation and prototyping in requirements validation.

(ii) Derive data flow diagrams and structure charts for the Unix editor.

(iii) Compare and contrast two different techniques of design validation.

(iv) Under what circumstances is a static program analyser most useful?

(v) Why is there no such thing as a self-documenting program?

(vi) Suggest how the working environment in the computer science department could be improved so that student programmer productivity is increased.

(vii) Describe the use of bar charts and activity charts for project scheduling.

APPENDIX II - Student Project Specification

The system description set out below is that issued to students taking our final-year undergraduate course in software engineering.

A Real-Time Patient Monitoring System

When a patient in hospital is seriously ill and taken to an intensive care unit, that patient's condition must be constantly monitored to check for deterioration and to alert staff if such deterioration occurs. Part of this monitoring can be done by machine with a display at the patient's bedside and, in some cases, this display is also fed to a central console where it can be watched over by a nurse.

It is common to monitor a patient's heartbeat automatically but other factors such as blood pressure, temperature and breathing are more often monitored by frequent manual checks. There is no technical reason, however, why these cannot be automatically monitored.

The aim of this project is to design a computerised monitoring system to collect the output from an array of instruments checking heart rate, temperature, blood pressure, and breathing rate, to record this information for future analysis, to display it in a convenient form both at the patient's bedside and at a central console, and to alert staff if potentially dangerous changes in the patient's condition are detected.

The following conditions apply to the system:

(i) It should be able to handle the monitoring of at least 8 patients and to display information about all of these patients at the same time.

(ii) Although initially it should be designed to support the monitoring of the parameters specified above, it should be capable of expansion so that further patient parameters may be added at a later date.

(iii) The system should provide different levels of warning ranging from 'take immediate action' for a cardiac arrest say to 'look when you have time' when a change in blood pressure, say, is detected.

(iv) The system should provide facilities to summarise the collected data about patients and to display and print these summaries in a convenient form for study by medical staff or inclusion in the patient's medical record.

It is an obvious requirement of such a system that reliability is of the utmost importance. The system therefore should include extensive self-checking facilities and should also be able to assess the feasibility of the information which is being presented to medical staff.

REFERENCES

{1} M.W. Alford, A Requirements Engineering Methodology for Real Time Processing Requirements, IEEE Trans. on Software Engineering, SE-3, no. 1, 1977, 60-69.

{2} K. Schoman and D.T. Ross, Structured Analysis for Requirements Definition. IEEE Trans. on Software Engineering, SE-3, no. 1, 1977, 6-15.

{3} R.C. Linger, H.D. Mills, and B.I. Witt, Structured Programming - Theory and Practice, Addison Wesley, Reading, Mass., 1977.

{4} L.L. Constantine and E. Yourdon Structured Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

{5} O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, New York, 1972.

{6} E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

{7} J. Alvey, Programme for Advanced Information Technology, HMSO, London, 1983.

{8} P. Freeman and A.I. Wasserman, Software Engineering Education: Needs and Objectives, Springer Verlag, Berlin, 1976.

{9} R.E. Fairley, Towards Model Curricula in Software Engineering. Proc. 9th SIGCSE Technical Symposium on Computer Science Education, Pittsburgh, Penn., 1978.

{10} H.D. Mills, Software Engineering Education. Proc. IEEE, vol. 68, no. 9, 1980, 1158-62.

{11} ACM Committee 1979, Curriculum 78, Comm. ACM, vol. 22, no. 3, 1979, 147-66.

{12} I. Sommerville, Software Engineering, Addison Wesley, London, 1982.

{13} R.W. Jensen, C.C. Tonies and W.I. Fletcher, A Proposed 4 year Software Engineering Curriculum, Proc. 9th SIGCSE Technical Symposium on Computer Science Education, Pittsburgh, Penn., 1978.