9

Marketing programming to nonprogrammers

Peter Juliff
School of Management Information Systems
Faculty of Business and Law, Deakin University, Burwood VIC 3125 Australia, e-mail: pjuliff@deakin.edu.au

Abstract

Based on a case study this paper looks at the problems associated with stimulating the involvement of noninformatics students in academic units related to software development. It provides advice, drawn from experience, on the means by which such students can be attracted to programming, can achieve a professional level of competence and can be encouraged to pursue further information technology studies.

Keywords

Programming languages, information bases, interaction and presentation, economics and business administration, noninformatics majors, curriculum (start), curriculum (core)

1 CASE STUDY AS BACKGROUND

Context

This paper draws on the lessons learned in the running of the Bachelor of Commerce (Business Computing) programme at Deakin University. Deakin is situated in the state of Victoria in Australia and has an undergraduate enrolment of some 26.000 students, half of whom are studying in distance education mode. The Bachelor of Commerce is the generic degree offered by the Faculty of Business and Law and has an enrolment of approximately 6000 students distributed over three campuses. Five years ago Deakin amalgamated with Victoria College. This was one

Informatics in Higher Education F. Mulder & T. van Weerts (Eds.)

in a series of such amalgamations in which Colleges/Institutes of Advanced Education (polytechnics) were combined with existing universities to form larger institutions and eliminate the so-called 'binary' education system which had existed up to that time.

Deakin IT studies and subjects

At Deakin, prior to this amalgamation, business (and any other noncomputing) students who wished to take information technology (IT) studies were obliged to do so by enrolling in subjects offered by the Department of Computing and Mathematics. The Commerce degree had a compulsory 'Introduction to computing' subject which was, of course, taught by the computer scientists. The Computing department offered a small number of IT subjects, including programming, which it hoped would attract commerce students beyond their compulsory first year subject. The enrolment in these units was negligible. When asked about the reasons for the reluctance to progress to further IT studies, the commerce students gave answers such as:

- 'it all sounds like rocket science';
- 'they put us in with the computer science students and then only lecture to them';
- 'we can't see any reason to have to study programming when we are going into a business career'.

Victoria College IT stream

At Victoria College, prior to the amalgamation, there was a viable computing stream in the Bachelor of Business degree which had a reputation for producing excellent commercial IT graduates in the traditional areas of systems analysis and design, software development and database management. These studies ran through the three years of the degree and attracted a large number of noncomputing students into the mainstream computing subjects as well as specialist units designed with such students in mind.

Contrasting scenarios

The amalgamation of the two institutions brought these two contrasting scenarios into conflict. The first act of the computer scientists was to contend that all computing studies of any kind within the amalgamated institution should be taught by their department. In this they failed and two separate IT academic departments were established: the School of Computing and Mathematics in the Faculty of Science and Technology and the School of Management Information Systems in the Faculty of Management - later renamed the Faculty of Business and Law. Having lost the first round the computer scientists then claimed that the teaching of programming was the exclusive province of computer scientists. This they won. By executive fiat the university decided that the School of Management Information Systems was to discontinue its Software Development major programme and that, henceforth, any commerce student wishing to study software must be taught within

computer science subjects. This meant a reversion to the prior Deakin model which had demonstrably been a dismal failure over the years prior to amalgamation.

This paper is the story of how, two years later, the School of Management Information Systems has more students enrolled in software development units than the School of Computing and Mathematics and attracts a large number of computer science students who follow the same units. The School of MIS does this without having a single 'programming' unit in its curriculum.

2 WHY TEACH PROGRAMMING TO NONINFORMATICS STUDENTS?

There are several reasons why students who do not necessarily see themselves as being computer scientists or as someone in a similar technical role, should seriously consider pursuing some IT studies. Not the least reason is that graduates often gravitate to positions which they never even contemplated during the tenure of their studies. Business graduates may well find themselves more involved in the technology of their occupation than in its commerce. Redmond-Pyle (1996) draws attention to changes in skill requirements of system developers and to the growing distinction between component builders, likely to remain computer science graduates, and solution providers who must understand the application domain. In this context of producing solution providers some of the major reasons for the teaching of software development skills are listed below.

To understand the nature of the operations of IT systems

By necessity the development of programs brings students into contact with all of the components of a computer system. It forces them to appreciate the tasks performed by operating systems and other of the more arcane components of a typical production system, such as utility programs, configuration control procedures, the functioning of internal memory and the nature of different data types and structures. All of these can be covered at a conceptual level in other subject areas, but the process of actually implementing a software system requires them to be dealt with at an operational level and, therefore, makes them harder to ignore.

To inject reality into other areas of the IT curriculum

One of the main problems with teaching the modus operandi of IT systems in subjects which are of the nature of analysis and design, is its presentation as a body of theoretical knowledge and skills which produces a blueprint from which an IT system will subsequently be implemented. Many of the problems in the design of any IT system, in the classroom or in the profession, only emerge in the process of the system's implementation. If students are required to take a design document and to actually bring the design to fruition as an operational system, they have the

opportunity to appreciate many of the problems which are unforeseen at design time.

The inherent nature of the task itself

I would direct readers to Fred Brooks' (1972) seminal work 'The Mythical Man-Month' in which he extols 'The Joys of the Craft' as being:

- 'the sheer joy of making things';
- 'the pleasure of making things which are useful to other people';
- 'the fascination of fashioning complex objects ... and watching them work';
- 'the joy of always learning';
- 'the delight in working in such a tractable medium'.

Brooks' statement that 'programming is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men' is as valid today as it was twenty-five years ago when the book was written.

To acquire skills in project management

Another of the most significant long-term problems in the implementation of IT systems is that of project management: the consequent difficulties in estimating the time for a software project and then monitoring the progress of that project. As always, these topics can be covered at a conceptual and descriptive level, but are much more readily appreciated and understood after the students have themselves been required to produce a software system in a specified time and with limited resources. This aspect of software development is best learned by having the students working in teams and reporting to a supervisor in an environment as close as possible to a working situation.

Even if the students do not themselves go on to be software developers, this aspect of their studies will equip them with an insight into the processes involved and better enable them to operate in a supervisory capacity if required.

The inherent training in problem solving

One of the most pervasive themes running through all IT studies is that of problem solving. Nowhere is this more essential that in the design and implementation of software. There are few other areas of human endeavour which are more insistent in their emphasis on the skills of problem analysis, the subsequent decomposition of large problems into their constituent smaller components and the rigorous specification of the interaction between those components. The skills acquired in the activity of algorithm design are transferable to almost all other facets of graduates' subsequent employment.

The computer is also a harsh mistress. There is a correct answer to a problem and the software either arrives at that solution or it does not. The program is therefore demonstrably right or wrong. In a descriptive discipline a student may be required to produce an 'acceptable' solution. In programming the student must produce a 'correct' solution.

The importance of integrating IT studies with non-IT disciplines
Given the all-pervasive nature of information technology in today's society it is difficult to imagine a career pursued by a graduate which does not involve a daily interaction with some aspect of IT within the working environment. The corollary to this is that the more comfortable graduates are with IT systems as a result of their undergraduate studies, the more opportunities will be open to them in their working life. If the study of software development can be made attractive to noncomputing as well as computing students, it provides an excellent vehicle to give all undergraduates a feeling for all of the design and implementation aspects of the type of software systems with which they will interact.

An appreciation of the complexities of human-computer interaction
An old proverb runs: 'I hear and I forget; I see and I remember; I do and I understand'. Given the availability of system development products such as Visual Basic and Delphi, it is not a difficult matter to have students put together a (simple) software system and to experiment with a variety of methods of presenting information on a screen and a variety of methods of soliciting interaction with a user. Despite lengthy lectures on the principles of graphical user interface (GUI) design, nothing brings home the problems of a clumsy human-computer interaction model like having to use it and demonstrate its operation to peers and supervisors.

3 WHAT IS THE BEST WAY TO TEACH PROGRAMMING?

There is an advertisement on Australian television for a breakfast cereal which has as one of its thematic lines: 'If you don't tell them that it's good for them, they'll eat it by the boxfull'. I have come to believe that this is the essential theme for marketing programming to noninformatics students. The surest way to alienate students who are not primarily enrolled for computer science curricula (and even some of those who are), from software subjects is to devise and describe a subject in the following style:

> **Computer Programming 101**
> A detailed study of algorithm design; multi-level decomposition; data typing and scope rules; logic constructs enabling structured programming and information hiding; abstract data structures and recursive processing techniques.

All but the committed student read these words and immediately think 'this is rocket science'.

Thirty years ago most of us who were teaching programming, were starting at the level of machine code or assembler language. When the students had mastered the intricacies of instruction addressing modes, indirect addresses, the binary representation of mantissas and exponents, the fetch-decode-execute cycle and the

conversion of relative to absolute addresses, we would allow them to move to a compiler language such as Algol, FORTRAN, COBOL or PL/I. While we may harbour a belief that this is still the way it should be done, the students do not believe us and they will not enrol for the subjects. The major factors in attracting noninformatics students to software development subjects are listed below.

Make the syllabus sound relevant, interesting and, above all, achievable

Compare the following unit description with the one above:

> **Systems Implementation 101**
>
> The aim of this subject is to develop computer based information systems which run in a Windows environment and have the same professional look and feel as other contemporary applications. You will become familiar with the techniques needed to design screens which interact with their users, including the use of the mouse, drop-down menus, message boxes and command buttons. At the end of this subject you will have a fully executing commercial computer application which will run on any PC and which may be demonstrated to a potential user (or employer).

Note that the subject is not called 'programming', although students will be learning to write programs. It makes no mention of logic constructs or data typing and scope rules, yet students will learn all of these. It emphasizes the point that the outcome is the production of software products similar to those in daily use in the business environment.

Provide students with instant gratification

The scenario of thirty years ago which was mentioned above, also included compilation turnaround times measured in hours, if not days. It was not uncommon to have a week elapse between tests of an executable program. With the current PC environment students can develop a system producing a procedure at a time with instantaneous compilation or interpretation, and testing. At the end of a two-hour tutorial a student can emerge with a small yet complete software application which may then be expanded and enhanced into an impressive product. At the end of a day at kindergarten, youngsters like to have a painting to take home to show the family and to have it displayed on the door of the refrigerator. What makes us think that undergraduates are any different?

Do not destroy the students' spontaneity

We are all aware that there are rules to follow in writing quality software. There are two ends of a continuum in the methods used in teaching programming to students. At one end there is the approach which, allowing students to write sloppy code, instils bad habits that may never be eradicated. This approach dictates that every program, no matter how trivial, must be written with the same attention to the precepts of good software design as a major safety-critical application. Time has led me to believe that this is a mistake. Students are so terrified of incurring the

lecturer's wrath over poor style that all of the enjoyment of just solving the problem is destroyed.

The other end of the continuum is a 'laissez faire' approach to the writing of small programs. A 'Nike methodology': just do it! Concentrate on the problem and the excitement of arriving at a working solution. When the students then realize that this is achievable, they can be convinced that a lack of discipline which was not overly important in a 20 line program would be a disaster in a 200 or 2000 line program. When they realize that programming is something with which they can cope and which they can enjoy, they will be prepared to learn how to do it properly.

Produce software with user interaction and a professional look and feel

How many software assignments have we generated for students which involved the construction and manipulation of complex internal data structures and involved little or no user interaction? While we may be convinced of the inherent value of being able to update a number of complex master files with a transaction string which needs conversion from variable to fixed field format using a state transition table to direct the logic, or the implementation of a depth-first search of a tree structure, the problem is that the students write a lot of code and see almost nothing happening for their efforts. We expect them to achieve their gratification from the knowledge that the job was done correctly.

This is not the type of motivation likely to appeal to noninformatics students. Such students need to feel that they are constructing software which is like the application software which they will encounter in their chosen discipline. They are not interested in the internals so much as the interaction with the professional user. The internal operations must be communicated subliminally. If we can attract them with the promise of relevance, we have the chance of extending their horizons to further, perhaps less inherently interesting areas once they realize that they can achieve in this field of endeavour.

4 WHAT TO USE AS A VEHICLE FOR SOFTWARE DEVELOPMENT?

First programming language

An issue which is guaranteed to generate a debate among teachers of programming, is that of the first programming language. Over the past thirty years I have used as an introductory language: machine code, Ecole, assembler languages, FORTRAN, BASIC, Pascal, COBOL, Scheme and, most recently, Visual Basic. I can name other institutions which are using Turing, Modula 2, Miranda, C or Delphi for this introduction. The majority of academic institutions in recent years have used Pascal or C as reported by Redmond-Pyle (1996), Jones and Pearson (1993), Morton and Norgaard (1993) and Furber (1992).

Does it matter?

Is it any more important than which car you first learn to drive? I believe that the answer is that it does matter, particularly to noninformatics students. Regardless of the merits of various models of cars I do not believe that any of us would advocate using a semi-trailer or a formula-one racing car to teach a youngster to drive. These vehicles - like some of the introductory programming languages used - require too much expertise even for the simplest of operations and are not representative of what 90% of the exercise is about. For those advocating the use of object-oriented languages as initial learning tools, Lee and Pennington (1994) point out the difficulty normally experienced by novices in coming to grips with O-O techniques and the likely resultant clouding of the experience of learning to program.

Visual Basic and beyond

Cox and Clark (1992) argue convincingly for Visual Basic as a first language due to its ability to be application oriented rather than syntax-driven. The choice to use Visual Basic as the introductory programming language in Deakin's business computing degree has resulted in an increase of software development enrolments from a total of around 100 two years ago to approximately 500 in 1997. And equally important, this increase is resulting in a commensurate flow-on of students into other information systems units. It is also attracting a large number of computer science students who are looking to increase their skills and exposure to GUI/Windows programming.

Having introduced students to software development via Visual Basic, they are led on to database design by writing applications which require them to interact with and update Access databases, and to further programming using COBOL for the batch processing of files which the students have created using their Visual Basic applications as the medium for data input.

At the end of this two semester unit software sequence, the students have developed and implemented applications which:

- use Windows GUI-applications and require an understanding of the methodology of designing event-driven software;
- use logic/procedure-driven software requiring a structured design methodology;
- manipulate relational databases;
- update commercially oriented files of indexed and random organizations;
- create a hybrid system using mutually acceptable file and data structures to communicate between programs originally written in different languages;
- are developed in teams using professional documentation standards and project management practices;
- have a professional look-and-feel similar to marketplace software likely to be encountered in the students' working environment.

It must also be remembered that most of the students would not consider themselves primarily as 'computing' students.

The attraction of Visual Basic is that:

- much of the activity in an application can be achieved by writing very few lines of code;
- the syntax is simple and intuitive while still enabling the teaching of rigorous software style;
- the programming environment is easy to use and enables the rapid development of applications which contain all the features of contemporary applications;
- it provides a bridge to other Microsoft products, providing students with a transportable skill;
- not the least importantly: it is a recognized skill in the employment market.

5 CONCLUSION

The essential aspects of marketing programming to nonprogrammers lie in making the task enjoyable, obviously achievable and professionally relevant. To this end those of us who are pursuing this goal must choose a delivery vehicle and applications which achieve a balance. On the one hand encouraging a sufficiently rigorous approach to software development so as to satisfy our own professional standards. And on the other hand allowing students some spontaneity in the exercise and their recognition that what they are acquiring are useful knowledge and skills which will be relevant to their chosen, noncomputing career.

6 REFERENCES

Brooks, F.P. Jr (1972) *The Mythical Man-Month.* Addison-Wesley, Reading, Massachusetts.

Cox, K. and Clark, D. (1994) Computing modules that empower students. *Computers and Education*, **23** (4), 277-284.

Furber, D. (1992) A survey of teaching programming to computing undergraduates in UK universities and polytechnics. *Computer Journal*, **35**, 550-553.

Jones, J. and Pearson, E. (1993) An informal survey of initial teaching languages in UK university departments of computer science. *University Computing*, **15**, 54-57.

Lee, A. and Pennington, N. (1994) The effects of paradigm on cognitive activities in design. *International Journal of Human-Computer Studies*, **40**, 577-601.

Morton, L. and Norgaard, N. (1993) A survey of programming languages in CS programs. *SIGCSE Bulletin*, **25** (2), 9-11.

Redmond-Pyle, D. (1996) Software development methods and tools: some trends and issues. *Software Engineering Journal*, March 1996, 99-103.

7 BIOGRAPHY

Peter Juliff is professor of management information systems at Deakin University, Australia. Immediately prior to this, he was head of the Department of Software Development at Monash University and has held several other senior academic appointments. He has spent over 30 years as an IT academic and practitioner, is the author of several books on computer science and software design and has conducted IT programs in Singapore, Malaysia and China. He is a Fellow of the Australian Computer Society and its chief examiner. He is the chair of IFIP Working Group 3.4 on vocational and professional IT education.