Has the Pascal Experiment Failed ?
or
Can A Good Language make Good Programmers ?

David Lowe, John Leaney
Computer Systems Engineering,
School of Electrical Engineering,
University of Technology, Sydney

Abstract

In the early 1980's the authors electrical engineering school switched their initial teaching language from Fortran to Pascal. One of the primary factors leading to this choice was the belief that a well-structured language such as Pascal would inherently lead to the students writing, as a matter of course, well-structured code. The students learning Pascal are separated into two degree courses, electrical engineering (EE) and computer systems engineering (CSE). The EE students complete only basic computing and software engineering material, whereas the CSE students cover a much more intensive software engineering curriculum. Experience over the last decade has shown that the EE students are not capable of consistently writing good code once they have completed the software subjects, whereas the CSE students are significantly more capable. It has become progressively apparent that the EE students cannot be forced into writing good code by restricting the scope of their tools. Rather they need to be explicitly taught good programming techniques. This paper discusses this shortcoming, its reasons, and possible solutions.

1 Introduction

This paper discusses the use of Pascal as a teaching language within both an undergraduate electrical engineering (EE) degree course and undergraduate computer systems engineering (CSE) degree course. This language was chosen due it well-structured syntax. It was believed that the use of a language such as Pascal would illustrate to the students a correct (i.e. structured) approach to software design and implementation, and hence lead to the students adopting such an approach. Pascal was believed to lead to improved programming and debugging skills [4, 3]. Pascal has now been taught for over a decade in the authors department, and analysis of the software development abilities of the EE students has shown that the above hypothesis is not supported. This problem is not encountered with the CSE students, who are taught a significantly higher degree of software engineering, making explicit many of the principles that were thought to be embedded in the teaching of Pascal. The following section discusses the use of Pascal as the first teaching language and the justifications for adopting it. The next section discusses the reasons for concluding that the experiment (i.e. the use of Pascal) has failed; using student abilities as the primary source of data. Possible reasons for this failure are then proposed and analysed. A series of conclusions that can be drawn from this experience are then outlined, followed by a series of suggestions for avoiding the recurrence of the problem.

2 Background

The authors belong to Computer Systems Engineering (CSE) group within the School of Electrical Engineering at the University of Technology, Sydney. In 1982 the school decided to switch its initial teaching language from Fortran to Pascal. Since then the degree course has undergone a number of restructurings and has spawned a second degree course (Computer Systems Engineering). Throughout these changes the initial language has remained Pascal and the method of teaching it has essentially remained the same, although the material being

taught has evolved significantly.

When it was first introduced (in a subject called "Computer Programming") it was a 3 hours per week, 16 teaching weeks subject. Along with the language syntax, only minimal material on structured design was included (such as the use of top-down design, structure charts, software life cycles etc). Progressively more material on software engineering has been added; data flow diagrams and data dictionaries, pre and post conditions, coupling and cohesion, Abstract Data Types. Additionally the use of the language was expanded to cover data structures; queues, linked list, stacks etc. At present the language and the initial software engineering principles are taught in two subjects: "Fundamentals of Computing" and "Software Engineering 1" (each is 3 hours per week, 14 teaching weeks). The details of these subjects can be found in [5] and are summarised in table 1.

In 1986, a second degree course (CSE) was introduced. The curriculum for this subject includes the same material as outlined above (as for the EE course) but then continues to develop this material in much greater depth. This includes a subject called "Software Engineering 2" (a 3 hours per week, 14 teaching weeks subject). This is described in table 1.

The justifications for using Pascal were many. Firstly it was a language which was readily available on the various systems which were to be used. Secondly, it was quite simple in its syntax, making it a relatively easy language to teach (and learn). Thirdly, and most importantly, it was a well-structured language.

Obviously good programming techniques and habits are desirable, and thus a language which promotes this is to be preferred over alternatives. At the time when the decision to switch to Pascal was made, structured programming techniques were considered to be the best alternative. Pascal is a language which actively promotes structured programming techniques. It was therefore hypothesised that the use of Pascal would tend to reinforce structured programming techniques to the students. This would be achieved primarily through Pascal illustrating the benefits and methods of structured programming

3 Failure of the Experiment

Having selected Pascal as the initial teaching language within the Electrical Engineering degree (and later also the Computer Systems Engineering degree) it was used for over a decade. During this time it has become increasingly apparent that the level of programming proficiency attained by the EE students has in general not been satisfactory.

In order to assess the success or otherwise of the use of Pascal in teaching structured programming techniques we need to have some method of assessing the quality of the software that the students produce. This is not a simple issue. It is not sufficient to simply look at student grades in subjects or assessment tasks involving programming. This is because the resultant grades will be highly dependent upon the difficulty of the particular assessment task. Similarly, the students cannot be compared to students in other courses (where a different approach has been taken) as there is difficulty in comparing the results obtained using different assessment tasks.

There are however two methods of assessing the quality of the software that is produced by the students. The first of these is purely subjective. It has come to be recognised by most members of the school involved in subjects with a software component that the electrical engineering students development of software is unsatisfactory. This is in general based not on the actual code that is produced, nor the look-and-feel of the resultant programs, but rather on the approach that is adopted in the design of the software. It is expected that the students should adopt a formal, well-structured systems approach to the design of the software (specification, design, implementation, testing etc). In many cases there is little evidence of this occurring. The software regularly has little or no evidence of correct design methods, requirements analysis, testing or supporting documentation. The students seem to be unwilling to utilise the formal methods being taught; they are being put off the use of data flow diagrams, abstract data types and other formal techniques. Rather the software that is being produced appears to be developed directly from the coding stage, bypassing most, if not all design. Other problems include the lack of testing and maintenance. One comment made by a member of staff when queried about the students weaknesses was "Students do not verify the accuracy of their results or even check to see if their result has any relevance ...". The only occassions when evidence of a satisfactory approach can be regularly be seen are when the assessment explicitly covers the analysis, design and testing, as well as the end-result.

A second, though also subjective, assessment of the software quality has become apparent since the introduction of the CSE degree. Students undertaking this course are given significantly more exposure to software engineering methods than the electrical engineering students. It has become apparent that there exists a disparity between the approaches taken by the CSE students and the EE students. It can be readily seen that the CSE students adopt a more thorough and formal approach to the design of software than the EE students irrespective of whether the assessment of the particular task explicitly covers the design.

Fundamentals of Computing (EE and CSE)	
Objectives	The subject aims to introduce students to computers, computer-based systems and computer programming in engineering environments, and to provide them with tools and techniques which will produce systems which are correct and maintainable. The students objectives are to learn the software engineering principles of analysis, design, implementation and testing of computer software. To become conversant with a programming language which encourages the use of software engineering principles. To use these skills and knowledge in a consistent methodology for practical software development.
Content	The subject is divided into 3 modules; (1) Introduction to computing and basic Pascal, (2) Structured analysis, design and testing, (3) More Pascal commands and data structures. The assessment includes group programming assignments.

Software Engineering 1 (EE and CSE)	
Objectives	To develop software development skills by giving the student an understanding about formal design in terms of data abstraction as a lead on Fundamentals of Computing. To understand the necessity of specifications in professional software engineering as opposed to software programming or coding. To be able to write rigorous specifications using data abstraction techniques.
Content	Implementation techniques (modules, pointers, recursion, OO concepts), data structures and algorithms (queues, stacks, lists, trees, sorting, searching). The course develops ideas and evaluation methods for module quality in the first module. The second module develops the abstract data type as a method of achieving high module quality with mathematical rigour. The third module proves the standard data structures and algorithms, formulated as ADT's. The assessment includes group assignments.

Software Engineering 2 (CSE)	
Objectives	To bring students to the point where they are fluent in the issues and objectives of software engineering, and competent in techniques to realise a set of requirements, apply rigourous software analysis, design, coding and testing procedures; understand and use object-oriented analysis, design, coding and testing techniques. On completion of the subject the students will be competent to engineer moderately complex engineering software systems, as members of a software development team.
Content	Software life cycle: life cycle models (waterfall, evolutionary, prototyping,); software estimating and costing; configuration management and requirements traceability; software maintenance; software reliability; software standards. User requirements: requirements principles and analysis; formal methods (pre/post conditions, ADT's, Z, VDM), DFD annotations. Class/Object Oriented techniques: analysis, design, coding and testing; inheritence, genericity, overloading. The methods used include team assignments with formal reviews between project stages, CASE tools, general engineering methodology applied to software projects.

Table 1: *EE and CSE Software Subjects*

4 Analysis

It has been shown that there are good reasons to believe that the "Pascal Experiment" has failed. In order to avoid a similar occurrence again the reasons for this failure need to be understood. If the causes are not fully comprehended then we can neither correct the problem with the teaching of Pascal, or stop the same problem from occurring in the teaching of alternative languages or paradigms.

In order to determine the causes we need to analyse exactly where the problem occurs. As was mentioned in the previous section, there are two main groups of students being taught Pascal; Electrical Engineering students and Computer Systems Engineering students. These students do identical initial computing subjects. The divergence occurs when the CSE students study subjects with a much a greater focus on software engineering, and its techniques, purposes and rationalisation. It has also been seen that the CSE students also appear to be better coders. Is this because of the additional material they are taught, or is it because this material impacts on what they were taught in the early subjects ? Irrespective of the cause, the existence of the difference indicates that the poor programming skills are not inherent, but rather it is certainly possible for the students to learn to write good software. It can therefore be concluded that the production of poor software developers within EE (as opposed to CSE) must be a result of the approach that is taken in the subjects that are completed only by the EE students. These subjects are those where Pascal and introductory software engineering principles are taught.

Two main hypotheses have been raised to account for the failure of the Pascal Experiment; the playpen hypothesis and the crafting hypothesis.

4.1 The Playpen Hypothesis

The first hypothesis is the *playpen* hypothesis. This is based around the recognition that a structured language such as Pascal essentially acts to limit the choices of the programmer. The language restricts errors (by, for example, type checking) and therefore forces the programmer to write code which is structured within the bounds of the language, and therefore presumably more correct (assuming that the language bounds are appropriately correct). In a learning environment, the students become familiar with correct code, and therefore will become more likely to produce this type of code. This proposition breaks down however in the light of the results that we have observed. A number of explanations can be put forward for this. The first is that the students are not allowed to learn from their mistakes, so that when they come to use another (possibly less-structured) language they are unaware of possible pitfalls. The students need to be able to make mistakes in order to understand why structured programming is necessary. By placing the student inside secure boundaries, Pascal does not allow them to comprehend the need for correct programming habits, or to place these needs into a broader context. Another possibility is that the structured methods used for Pascal may not be directly translatable to other structured languages, though this is less likely.

The playpen hypothesis can only partly explain the observed failure. This is because it is aimed mainly at the consequences once the students are allowed out of the "playpen"; i.e. they graduate to more complex, or less-structured languages, where they can make mistakes which Pascal prevented. We have found however that often the quality of the programming was poor even when the students continued using Pascal (i.e. they were still within the playpen). In order to explain this we need to consider an alternative hypothesis: the crafting hypothesis.

4.2 The Crafting Hypothesis

The crafting hypothesis essentially states that learning to become a good programmer is not about solely about adopting the techniques that are made explicit in a language such as Pascal. There are concepts and/or techniques which are neither explicit nor implicit in the language, but rather need to be taught separately from the language. These concepts have (at this stage, somewhat arbitrarily) been grouped under the title *crafting*. This implies that crafting needs to be taught independently of the language syntax and semantics.

We need to initially consider what concepts are not inherent in the language, and therefore what needs to be covered separately. We then need to determine whether this is currently being covered, and in sufficient detail. At present, as stated previously, the introductory computing subjects contain the material detailed in table 1. Obviously there must be material which is missing from these two initial subjects, which is resulting in the students failure to produce acceptable software.

An indication of the material which is missing can be obtained by considering that the CSE students do become good software engineers. They initially cover the same material as the EE students and then continue with greater detail. The additional material that they cover must lead to the differences that have been observed. It is important to realise that it may not be practical to teach the EE students the same material in order to make them good programmers, nor may it be necessary (much of the CSE material will be irrelevant to a typical EE graduate). It should however be able to give an indication of what can be done to rectify the situation.

The major differences between the material covered by EE and CSE falls into three categories:

1. Knowledge
2. Context
3. Experience

The CSE students are given a much greater knowledge regarding software engineering principles and techniques. The material does not however include any fundamentals which are not taught to the EE students. It would appear that the difference does not lie in the additional knowledge obtained by the CSE students. However the additional knowledge may well act to place the earlier material into context, thus given the CSE students a greater motivation to use the tools and techniques provided.

One final possibility may be that the CSE students have more exposure to software (and larger software systems) than the EE students. This will give them a greater experience, again acting to increase the students motivation to use the tools provided.

This tends to indicate that although we do not necessarily need to teach the EE students as much software engineering material as the CSE students[1], we should at least attempt to give the EE students greater motivation to use the correct engineering tools in the design of their software.

5 Solutions

In order to improve the situation from its present condition it is not sufficient to simply swap languages. We

[1] The exact amount of software engineering that should be taught to our EE students is an ongoing debate

need to place a stronger emphasis on the dual issues of firstly, placing software engineering into context so that the students can recognise the need for software engineering techniques, and secondly, the teaching of software crafting.

The first of these two issues gives the students justification for using software engineering techniques to move from a problem description to the software design using formal techniques rather than using a haphazard approach. Focussing on crafting will involve placing less emphasis on the particular language and more on the methodology of converting a design into code. Obviously the language syntax stills needs to be taught, but it should be possible to achieve this in such a way as to reinforce the crafting of code. Possible techniques to achieve this may be case studies (i.e. illustrate by example), problem-based learning, group assignments incorporating peer review (where formal techniques become more important) etc.

Placing software engineering into context is not a simple issue. Alder [1] describes software engineering education in an electrical engineering degree course (at Bradford University in the U.K.). He claims that "[A worthwhile software engineering education] can be achieved by including a significant element of practical, especially group-based, work". This may not always be feasible due to the restrictions on the amount of time that may be spent on any given topic. It can however be used as a guideline for including as much practical material as possible. Benenson [2] adopts a significantly more informal approach, primarily to illustrate the benefits of software engineering. This tends to place the techniques used into context without becoming distracted by the details of particular methods, emphasising their effectiveness, and making the students more likely to adopt them.

It has been concluded above that the problem does not lie in the language itself, but rather in the way in which the language is taught. Given this conclusion we can state that any language could be used as an initial language, and result in good software engineering practice, if it is taught properly. The real situation is however slightly more complicated than this first appears; different languages have syntaxes and semantics which are of varying levels of complexity. This will affect how simple the language is to comprehend. A complex language which is difficult to learn will distract from the concurrent process of learning correct software development and crafting. What we can say however is that the choice of language becomes less influenced by the language paradigm, and more influenced by other factors such as language (and environment and tools) availability, syntax complexity, availability of texts etc.

It may become apparent that for some courses Pascal still remains the preferred language, though this should be for the reasons just mentioned rather than because of a belief that its structure will force a structured software design methodology. Correct software engineering principles need to be explicitly taught.

6 Conclusions

Over a ten year period, we have attempted to teach Pascal in an undergraduate electrical engineering program. The primary reason for the selection of Pascal was that it enforced a structured approach to software development. Experience has shown that this is not true. What Pascal enforces is structured code, not structured coding, or structured software development. In order to teach students correct software development it needs to be taught explicitly.

The authors are currently involved in redesigning the software thread within both the EE and CSE undergraduate courses to correct the problems outlined in this paper. A significantly greater emphasis is to be placed on software crafting, in an attempt to produce students who are significantly more capable of understanding the need for software engineering and then acting on this understanding.

References

[1] C. Alder, "Software engineering education in an electronic engineering degree," *Software Engineering Journal*, pp. 191–199, July, 1989.

[2] G. Benenson, "Using software engineering to break the programming barrier," in *1988 Conference on Frontiers in Education*, pp. 394–398, 1988.

[3] R. E. Mayer, "The psychology of how novices learn computer programming," *ACM Computing Surveys*, vol. 13, no. 1, pp. 121–141, March, 1981.

[4] D. Stone, "The impact of Pascal education on debugging skill," *International Journal of Man-Machine Studies*, vol. 33, no. 1, pp. 81–95, July, 1990.

[5] University of Technology, Sydney; Faculty of Engineering, *School of Electrical Engineering Handbook*, 1993.