

Choosing languages to teach programming paradigms and programming styles

J. Traxler

*School of Computing and Information Technology,
University of Wolverhampton, Wulfruna Street,
Wolverhampton, WV1 1SB, UK*

Abstract

The comparison of different programming paradigms, styles and languages is a common topic in undergraduate computing courses. This paper looks at some of the factors involved in choosing programming languages that will support the effective comparison of language design issues and of different programming paradigms.

The paper is based on the experience of an undergraduate module in the computing programme at the University of Wolverhampton, Software Construction, run for final-year computer science and software engineering students at the University of Wolverhampton and draws on feedback from the students.

The conclusion is that, even in examining only a small number of new programming paradigms, using only a handful of languages, an effective understanding and comparison of languages and paradigms is difficult to achieve. This is partly due to the different approach each language adopts to issues of general language design, partly due to the effectiveness and purity with which a language will implement its chosen paradigm and partly due to the relatively small number of working languages representing the possible this wealth of different parameters. Effective comparison is impossible because there are too few languages to systematically compare "like with like".

These difficulties are compounded, in teaching these ideas, by the fact the previous exposure and loyalties of the students to languages are often very varied and also usually heavily skewed by their first or working language. Students' perceptions of the industrial credibility and employment potential of languages also comes across as an important determinant of their receptiveness.

1 Introduction

Whatever programming languages and paradigms are currently used by the educational and industrial mainstreams, we must recognise the need to give computer science and software engineering graduates sufficient technical breadth and background for their subsequent survival in a programming world characterised by increasing change, complexity and fragmentation. In particular we need to give them some idea of the diversity of programming paradigms, styles and languages.

292 Software Engineering in Higher Education

This is obviously the kind of contextual education that must be aimed at the final year of a degree course where we can consolidate and broaden the more practical programming modules in the earlier years of a course. Ideally we would like to give an even-handed overview of the paradigms, styles and philosophies but we must recognise that several factors will militate against this. Firstly, the previous programming backgrounds of the students, before and after they started our course, and secondly, the disparate and incommensurable attributes of the languages themselves. In choosing languages to use in this kind of teaching, it is necessary to identify these various attributes of programming languages and choose languages that clearly exemplify and isolate them.

This paper looks at a module, Software Construction, run at level 3 in the School of Computing at the University of Wolverhampton, and based on the feedback of the students themselves, tries to address some of the problems of this kind of comparative and contextual module. The verbatim text of the student feedback especially their expressed learning difficulties can be found in an earlier paper [1].

2 The Module

The module is now in its third year and has proved popular with undergraduates. It attempts to define some of the characteristics of programming language design and to review the procedural paradigm. It then introduces the theory and practice of the functional, object-oriented and concurrent paradigms, as well as referring briefly to assembler, database, formal methods and systems programming. The languages mentioned include Ada, Basic, C, C++, Modula-2, Miranda, occam, Pascal and Smalltalk.

The module was assessed by open-book exam, coursework essay and coursework programming task. The choice and availability of languages influences the nature of the assessments.

The strategy in the essay assessment was to make the students review and assess the factors that affect attempts to program in a particular language in a particular style. Students chose one language for each paradigm and discussed and compared surface features, facilities for programming-in-the-large and programming-in-the-small, implementational and practical aspects of the language and then assessed whether the language was a good implementation of its paradigm.

The strategy in the programming assessment was to present the students with simple problems that were amenable to solution by all three of the new paradigms and see if their answers managed to express the differences adequately. Setting these assessments, there seemed to be a relatively small repertoire of such problems : sorting, searching and other operations on lists, matrices or arrays and operations on data structures. In pursuing this approach, there were risks of smoothing out the distinctiveness of the paradigms and not giving any particular paradigm a chance to excel at its own sort of problem. Nevertheless we hoped that we would counter the impression that each new problem demanded a new language and get the students beyond writing "hello world" programs in more and more new languages.

3 Issues

Some of the issues that this module has raised have included :

- i. The extent to which students' previous experience and training skews their understanding of new programming paradigms and languages.
- ii. The extent to which the University's choice of base teaching language, initial programming paradigm and programming style affects the students' subsequent progress in other languages and paradigms.

Software Engineering in Higher Education 293

- iii. The relative advantages of using either "pure" languages (eg occam, Smalltalk, Miranda etc) or "hybrid" languages (eg C++, Modula_2 etc) to introduce and implement different programming paradigms.
- iv. The relative advantages in trying to use a general-purpose or universal language, such as Ada, to illustrate every paradigm.
- v. The difficulty students have in distinguishing between the surface features of a language, its environment, its typical problem-domain, its implementation, its "real-life" facilities (such as file-handling or graphics) and its style and culture.
- vi. The difficulty students have in reaching an understanding of the principles of a language whilst avoiding a detailed study of its syntax. Should we use a universal language, such as Ada, or lots of simple languages, such as occam and Miranda? Should we use small, academic languages or large, general languages?
- vii. The relation between the paradigms with their respective languages, on the one hand, and appropriate formal methods and analysis and design activities, on the other.

4 Students' Previous Exposure and Exposure

The students were asked to give an account of their programming history to give some insight into the breadth of their exposure to different languages.

It turned out that their initial programming paradigm was almost universally imperative. The majority of those with pre-university programming experience, about fourteen of the forty-odd, started programming at home or school using Basic, divided equally between BBC, ZX80, Commodore and unspecified dialects. Many of these, seven, went on to the corresponding assembler as their second programming language.

Some others had also gone on to 68000 or 8086 assembler as their second language.

Another group of about of nine had learned Pascal as their first language ie Pr1me Pascal as their first language, many of them on arrival at university. The rest of this group had learned Turbo Pascal when we changed over.

A few part-timers had gone onto Windows programming at work. A handful had C or Cobol as their first language; or had one or the other as their second. There was finally a smattering of just about everything else including Lisp and Ada (the French exchange students), Oracle, Informix, Fortran and Clipper.

Thus the first factor that we need to address in choosing our teaching languages, especially in a modular scheme, is the bias, the variety and the variability in students' prior experience.

5 Defining a Paradigm

"Paradigm" might sound a needlessly academic word but was fundamental to the module and not to be confused with, say, "style" nor to be taken to be synonymous with specific languages that is, we did not for example want students to assume that object-oriented programming always meant C++ and that C++ always meant object-oriented programming. Consequently, we used an operational definition that a paradigm was some framework for writing textbooks, setting exams, testing competence and defining new development targets and research objectives.

294 Software Engineering in Higher Education

6 Candidate Paradigms

The following are some of the more obvious programming paradigms that a comparative programming module could address : imperative or procedural programming, functional programming, declarative programming, concurrent programming, object-oriented programming, assembler programming, systems programming, database programming and formal methods (with reification). This is obviously far too many in the context of a 10-credit module, and we had also to assume that students had would meet some of the candidate paradigms elsewhere.

Consequently, we elected to concentrate on concurrent programming, functional programming and object-oriented programming as giving a representative insight into the idea of a paradigm and to use procedural programming to cover more general issues of language design.

We could then use some of the others to help discuss the idea of a paradigm : for example, were systems programming or assembler programming paradigms in their own right or just procedural programming in specific application domains or languages ?

7 Language Design

We felt that in teaching students about programming paradigms and the languages that implement them, we needed to discuss the issues that underpin the design of languages in general. We therefore needed to identify some languages that illustrated options in language design. Because the students were already familiar with procedural programming in some sort of Pascal, we chose this and then Modula-2 to illustrate attempts to improve Pascal; Ada as an exemplar of modern software engineering practice and C as the obvious industrial language. This choice usefully led to discussions on power and simplicity, style and culture and also standards and dialects.

It was often apparent that this was the first time students had had the opportunity to articulate their reactions to learning Pascal or C as a first language and also the first time that they had had the chance to see their difficulties in learning a language as a function of the language's (possibly) poor design rather than their weaknesses as learners. Moving on from this to proposing or evaluating alternatives, especially hypothetical ones, was less easy.

8 Surface Features

This phrase was intended to encompass those attributes of programs and programming languages that relate to the appearance and construction of programs as documents. The issues include capitalisation and character set, free and fixed formats, indentation and nesting conventions, flexibility of commenting, rules for identifiers etc.

The approach taken by language designers to surface features was one aspect of language design that was discussed early in the module. Procedural programming in Pascal, Ada and C provided a wealth of small accessible examples of the choices made by language designers in choosing surface features. Moving from description of surface features to any evaluation of them proved more difficult and students' coursework, which required some comparison and evaluation of languages, was generally full of descriptions of surface features at the expense of any evaluation. As many of the students compared Pascal with Modula-2, in some respects intended as an "improvement" by its designer, this is difficult to understand.

9 Generality, Power, Simplicity and Flexibility

These are related and contrasting attributes of languages and are an important part of language design.

Software Engineering in Higher Education 295

Simplicity is obviously just what it says and is an indicator that students can learn a language easily and practitioners can use and maintain it safely. It is not merely a function of the number of keywords but also the levels of precedence, the mapping between the intuitive meanings and the actual meanings of expressions, the extent of overloading and many other factors. Simplicity should obviously be an attribute of educational languages such as Pascal and Modula-2 and so students must become predisposed to expect it. Conversely they may also become ill-disposed to languages that lacked it.

Generality, in our definition, is the ability of a language to express solutions in a variety of ways, whilst power is the ability of a language to deal with substantial problems as well as simple ones. Because of its apparent generality, we tried initially to use Ada as a language to that would span as much of the teaching as possible. The reaction of students to the apparent size and complexity of Ada was very unfavourable.

There is a tension between simplicity on the one hand and power and generality on the other but in giving students a direct experience of that latter was latter.

10 Programming-in-the-Large and Programming-in-the-Small

These two attributes are two complementary aspects of the use of any programming language. The idea of "programming-in-the-small" is probably relatively easy to understand and discuss because it deals with the ability of the language syntax to support the construction of expressions and algorithms, by provision of appropriate control and data structures. In their coursework evaluating languages vis-a-vis programming-in-the-small, students were inclined to see more syntax as the best option, although they frequently admitted a preference for whatever resembled their own first language.

The phrase "programming-in-the-large" relates to the ability of the language to support multi-programmer, industrial-sized programs and as students generally learn and practise programming on a very modest scale, it is an attribute far harder to value when meeting new languages.

Fortunately (sic) most of the students had used monolithic classical Pascal and could see the virtues of Modula-2's modularity, especially when linked to reusable module libraries, but had trouble comparing the specifics of Modula-2 with Ada and C in this respect on the basis of a relatively superficial introduction.

11 Standards, Dialects and Portability

These features of language design were also slightly remote from students' first-hand experience. They could see the problems involved in learning a loosely-standardised language in the face of conflicting textbooks and compilers but also the attraction of using whichever dialect offered most support on their current project.

Few students, except one or two part-time students, had ever ported a program from one machine or compiler to another and so standards were not an issue to them. Many part-time students were using languages based on a "de facto" industry standard eg dBASE at work. They understood the value of their expertise and were prepared to look at the extent to which they were locked into a particular environment.

Given this relative inexperience, it was difficult in comparing languages to bring out the merits of standardised and machine-independent languages and less standardised or machine-specific languages, especially as the obvious competing exemplars were C and Pascal.

12 Paradigm, Style and Culture

Whilst we can usually identify what paradigm any given programming language is attempting to

296 Software Engineering in Higher Education

implement (though how well it does it is a separate and important question), there are nevertheless other related attributes of a programming language that less easy to define. These are its style and culture.

A language that can clearly illustrate ideas of style and culture is obviously C, especially when compared to say Modula-2. Both languages were designed primarily as systems programming languages and thus fall within the same paradigm, and both use similar techniques to reach similar objectives. But over-and-above these explicit differences, there are such wide discrepancies between how their respective programming communities expresses themselves that it must be valid to talk about programming style and programming culture as ideas that are arguably independent of programming paradigm.

13 Environments

Although the programming support environment, for example the apparent file structure, the editor, debugger and so on, are attributes of the implementation not the language itself, we cannot actually use any language without one and students often see them as more intimately linked than they should at this academic level. This makes comparison of, say, JPI Modula-2 with Borland C difficult because of their different environments. Obviously it would be attractive from the teaching and learning point-of-view if all languages could be accessed through the same environments. Families of languages such as Borland's or Microsoft's offer such uniform environments but minority languages such as occam, Modula-2 and Miranda will always be exceptions and in some cases, for example smalltalk, the environment may be an intrinsic part of the language. Likewise, it could be argued that occam's folding editor and Ada's APSE are integral to their respective languages.

14 Industrial Credibility

Our feedback from the students illustrated the importance of the industrial credibility of languages if the languages were to be taken seriously. This was a conservative influence but did not impair their enjoyment of Miranda, which they nevertheless saw as an academic irrelevance. It did, though, strongly influence their reaction to Ada and Modula-2, especially in any comparisons with C; likewise the reception of Smalltalk was prejudiced by the status of C++. The other factor influencing the reception of new paradigms and languages was a strong and conscious preference amongst for the language that they used first or most. This accounts for our choice of Modula-2 for concurrent programming and Turbo Pascal for object-oriented programming. Unfortunately, as long as we use industrially irrelevant languages such as Pascal as our base language, there will continue to be a tension between the two factors outlined above.

15 Design and Formal Methods

For a programming languages and programming paradigms module, it was important to try to establish the links between paradigms and languages, on the one hand, and some basic design methods and appropriate formality, on the other hand. This was to provide a context for programming and avoid the impression that languages and paradigms were purely technical and programming issues.

It would have been interesting to consider appropriate design methods and notations and formal methods that mapped well onto a language representative of each paradigm and it would have desirable if such topics could all have been treated to the same depth and in an easily comparable fashion. For example, Hoare's CSP could have been introduced in looking at concurrent programming in occam, the students having already studied a limited amount of Z with Pascal, or object-modelling diagrams could have been introduced as an object-oriented design tool, to compare with module hierarchy charts and PDL.

In practice the module was not big enough and reference to design methods was limited to the relatively informal. Thus it was all the more important that the relationship between the design and the language was intuitive.

16 Pure vs Hybrid

To a greater or lesser extent, all three paradigms were characterised by the availability of both "pure" and "hybrid" languages, that is languages that have implemented solely one paradigm from the outset as opposed to languages that have attempted to implement a new paradigm by grafting syntax onto a language intended for a previous paradigm. Arguably, this choice could be illustrated in concurrent programming by occam versus Modula-2 (though the distinction may only be one of granularity) and certainly in object-oriented programming by Smalltalk versus C++.

17 Object-Oriented Programming

Currently our base teaching language is Turbo Pascal but this was an interim choice between the P_rime Pascal used with procedural programming until recently [2] and something more permanent. The decision has now been taken to change the base teaching language to Borland C++ and object-oriented programming. Several new introductory modules will be on stream in 1994/1995. Because the Software Construction module is a level-three module in a very modular degree scheme, the effects of this change will be felt slowly over several years and in the meantime the students on Software Construction are likely to be very mixed in their backgrounds.

For the last two years we have introduced object-oriented programming using the object-oriented extensions to Turbo Pascal. This is an object-oriented hybrid language that all the students can quickly pick up from some background in Pascal and the principles involved also allow a brief introduction to C++ for the partisans of C.

Turbo Vision was used to show an object-oriented style of developing simple event-driven programs with minimal effort and the students were given a practical introduction to Smalltalk.

Of these various elements, Turbo Pascal and Turbo Vision were the best received whilst C++ was only interesting to C programmers in spite of its industrial significance. Turbo Vision was used successfully by several students in their final-year projects although the design approach adopted was rather intuitive. Eiffel was clearly "just another academic" and Smalltalk was conceptually hard work in spite of its current glamour.

18 Functional Programming

With functional programming, many of the choices are straightforward. In many teaching contexts, it is a new area uncluttered by hybrid languages or industrial interest. It also seems to have been historically an area of programming unsullied by language wars. The programming paradigm maps cleanly into a handful of similar languages, Miranda and SML being the most obvious. Purely for reasons of commercial availability we acquired Miranda on our Suns running under Unix. This has proved very acceptable to the students so we will consider the PC version as well when it becomes available.

In their reviews of functional programming and Miranda, the students have been appreciative of its simplicity at most levels, scathing about its commercial and industrial relevance and its ability to support industrial programming and threatened by its terseness, its resemblance to formal methods and its use of recursion.

19 Concurrent Programming

Concurrent programming has a relatively long history and because of its potential for solving operating system throughput problems, it has attracted serious academic and industrial interest. Attempts to explain

298 Software Engineering in Higher Education

and implement concurrency have often been based on relatively low-level constructs, constrained by the specifics of their hardware platform.

Much of the literature contains elements of operating systems theory and hardware constraints that make it difficult for beginners to understand. Even modern implementations can seem awkward, impure and imperfect. Although occam is new and pure, it is also rudimentary, research-based, fine-grained and tied to the transputer. Unix, with its forks and pipes, provides a far coarser-grained parallelism but only accessible to competent Unix systems programmers.

In addition, in teaching we were also confronted with the major issue of choosing between monitor-based or message-passing languages, or perhaps discussing both. Previous attempts to use Ada for illustrating general language design and for teaching concurrency on similar modules suggested that the volume of new syntax would prove tiresome whilst the similarities of Pascal to Modula-2 meant a fairly small syntactical overhead. So Modula_2 was the obvious candidate.

The University currently uses JPI Modula-2 with a locally developed kernel that supported processes and signals. Although ostensibly a highly standardised language with good introductory textbook support, Modula-2 in practice always seems weakest in the area of concurrency. Our local kernel is imperfectly documented and slightly idiosyncratic and proved more trouble to students than it rightly should. As the students had no difficulties with procedural programming in the JPI Modula-2, we assume their difficulties were specific to the local kernel. In every other respect, Modula-2 seemed an excellent choice for exercises of this scope.

The alternative facilities made available through Ada and occam were discussed through paper-based exercises.

20 Conclusions

Ideally any attempt to compare programming languages or paradigms should be based on identifying and isolating specific characteristics and examining and assessing their variation over different languages or paradigms. There should be enough varied languages to enable a comparison of "like with like".

It is clear from this informal survey of the teaching issues and the languages available that there are so many issues and only relatively few languages that this approach is not viable : one can instead only identify relevant points and hope to find the implementations of them in some sample languages. This risks causing confusion because students may not clearly pick up the point of the comparison.

The absence of a programming language capable of expressing several different paradigms means comparability across the paradigms is difficult to achieve. The languages we are forced to use differ in complexity, power, generality and size; in environments and programmer support.

Two major additional factors that influence the choice of languages in programming paradigms modules are the relative industrial credibilities of different languages and students' previous experience and exposure to languages, especially their initial teaching language.

21 References

1. Traxler J., "Teaching Programming Languages and Paradigms", Second All Ireland Conference on Curriculum and Computing, Dublin, 1994
2. Traxler J., "Factors that govern Initial Programming Language Choice", Open University Workshop on the Choice of Programming Languages, 1993, to be published as "Programming Languages : Practice and Experience", Chapman and Hall, 1995