Coherent Metaphor Sequences in the Teaching of Programming

Vicki Sivess & Hugh Glaser
Department of Electronics and Computer Science, The University of Southampton, Southampton, U.K. S017 1BJ

Abstract

At Southampton University, we introduced Standard ML into the first year programming course as a vehicle both for teaching programming and for introducing principles of software engineering. The paper first outlines the background of the students and the depth of programming experience prior to joining the course. The class was divided almost equally into those with experience equivalent to A level or BTEC diploma level computing, plus a few professional programmers, and those with very little or no experience. For half the class, then, this is not their first programming language, and such students arrive with existing models of the programming process. The paper looks at which software engineering principles the different parts of the language support. We then consider how the underlying themes and the metaphors implicit in the material cohere with what they already know and what they are learning at the same time.

1 Introduction

The Computer Science degrees at Southampton are marketed as having a strong Software Engineering flavour. When we introduced a functional language into the first year, in October 1991, we chose Standard ML (SML), on the grounds that it would serve as a vehicle for introducing principles of software engineering. We felt that it had many features that made it suitable for this purpose and that, additionally, it was considered a candidate for serious implementation work by industry [1]. The language is the first one that Computer Science students are introduced to at Southampton. It is taught for the first six weeks of their course. Each week, a student attends three hours of lectures, one small group session and one two hour laboratory session. A weekly coursework is set and returned the following day to give immediate feedback on progress. They also carry out one

larger assignment. Students are given booklets containing all the notes and laboratory material at the beginning of the course. The material is available online. Students are encouraged to become active participants in the course by using logbooks, doing background reading, trying out examples, and so forth.

In this paper, we first outline the background of the students and consider some of the metaphors first years already bring to the task of learning to program in SML. We look at which software engineering principles the different parts of the language support. We then look at what metaphors are implicit in the material learned and how they relate to what the students already know and what they are learning at the same time and we draw some conclusions about our experience.

2 The Background of the Students

For 1993/4 we had an intake of seventy five. The standard entry requirement is 22 points at GCE Advanced level, with a grade C in mathematics, or equivalent qualifications. Twenty five students (one third) have non-A level qualifications. Twenty nine students have A level or a BTEC diploma in Computing. If we add to this number those students who in other ways have had plenty of computing experience (e.g. they are mature students who have been programming in industry) then we can say that about half the class had done a great deal of programming while the rest of the class had done some, little or none. While the course is necessarily taught assuming no experience, in reality, for half of the class, this is at least their second programming language.

This profile has changed over the last few years: the class size has increased; the proportion of non-A level candidates has increased; the proportion of first year students with substantial prior knowledge of computing has increased and we are presently relaxing our requirement for mathematics. While this leads to a richer variety of experience amongst our first years, it means we can no longer rely on the fact that we are teaching a small homogeneous group with similar backgrounds, and this presents us with new challenges.

3 Metaphors

A simile is when we say one thing is *like* another. For instance, we say "My love is like a red red rose". A metaphor is when we say one thing *actually* is another thing. We tend to think that it is rather a poetical way of expressing ourselves and it is, indeed, a device used in poetry. Closer study reveals that our language systems are built upon metaphors to the extent that we simply do not even notice them. Lakoff and Johnson [2] have claimed that they are so pervasive that our whole conceptual system is largely metaphorical and by using metaphor we understand new experiences in terms of things we already know. We say, for instance, "I demolished his argument." and Lakoff and Johnson claim that this is part of a whole *coherent system* whereby we talk of arguments in terms of war.

Students entering our degree course therefore come armed with all the metaphors common to our culture, including those relating to machines and, specifically, computers. This will include metaphors gained through their use as games machines, for example, where they become instruments of control, or as spreadsheets or wordprocessors, where the notion of a *dialogue* may be central [3]. Their reading in the area of science fiction may well have suggested the idea that machines are intelligent, and many will be aware of the Internet and the way it is talked about in terms of cyberspace. Those students who have already done some programming will have built up their own sets of models and metaphors about the process that is involved.

4 Putting the Course into Context

During the first lecture (and in the course booklet), we say

> Teaching you to program well is part of the process of equipping you to become competent software engineers which we hope you will be by the end of the degree.

We thus make explicit that what the students learn on this course is intended to provide the start of that process and will, by implication, introduce them to some necessary foundational concepts. It is not meant to complete the process, however. It also positions the course as part of a cohesive three year programme.

In the same lecture, we outline some of the key quality requirements which users have from a software system. These were influenced by the five quality areas defined by ICL's OPEN*framework* architecture [4]. We say

- These are some attributes that a customer in industry expects to find in software:
  - it gives correct answers
  - it helps in the running of the business
  - it is available when needed
  - it is pleasant to use
  - it is secure from malicious or accidental damage
  - when the business changes or the customer has new requirements, the software can be changed accordingly.
- Many of these attributes can be obtained by building quality software according to good engineering practice.
- Here are some of the sound engineering skills you will learn during the programming course:

- to write correct, maintainable and reliable software
- to specify and design programs using abstraction techniques
- to use programming techniques that are amenable to formal verification
- to produce reusable software.

4.1 SML and Software Engineering

SML has features that encouraged us to think that it would enable us to teach the students to build quality software according to good engineering practice. SML is a strongly typed language with polymorphism and type inferencing. It supports pattern matching in argument strings. Structures and functors provide a very powerful generic modules system. In this section we outline the content of each lecture and emphasize the software engineering principles covered.

Week one

In the first week, types are presented as a way of logically organising data and functions as a prime abstraction mechanism. Local declarations keep the environment simple and thus easy to understand and reason about and the modules system is a calculus of environments. The ability to abstract away from the details of how a function is built and consider only the external behaviour of the function when using it gives us an encapsulation mechanism for building large systems. Ideas of reuse and top-down refinement are introduced.

Week two

Conditional expressions in functions and pattern matching in arguments support case analysis as a problem solving strategy. Lists and tuples introduce the idea of structuring data in a way that matches reality.

Week three

We develop further the ideas of functional decomposition and problem solving. We show how subexpressions in our program correspond to the solutions to subproblems, and the program comprising these subexpressions corresponds to a complete problem solution. We introduce the idea of robustness in programs by showing that some applications of a function do not make much sense in terms of our problem domain and that, at present, the function does not differentiate between correct and erroneous input, provided it is of the correct type. Polymorphism and overloading allow us to introduce the idea of generalising functions and reuse of components. An example shows the need to specify carefully what should happen in certain cases, before implementing a function. We comment

on the implementation and see that for some operations, a different data representation would have been more appropriate.

Week four

Higher order functions make it possible to define very general functions that are useful in a variety of applications. Because they are usually polymorphic, they allow us to capture a whole set of particular functions in one definition. They are very useful for prototyping. We talk about writing programs in a way that copes with complexity, particularly with respect to understanding a program text and maintaining it. The pattern of connectivity between components is at the heart of reducing complexity. Concepts of cohesion and coupling are introduced.

Week five

We talk about defining concrete types but lay the groundwork for the introduction of abstract data types. Recursive type definitions and new type operators are introduced. We talk about building correct and robust software. We say that robust software behaves sensibly under adverse conditions and show various approaches to this. We introduce the exception handling mechanism of SML.

Week six

We try to make the students aware of the scale of some larger programs and present the modules system as a facility to support the incremental development of large programs. We introduce information hiding and abstract data types. We show how to replace one implementation with another. We discuss the use of signatures to develop programs, enforcing the idea of postponing implementation decisions for as long as possible. We develop an abstraction in a top down fashion by writing an equational specification. We show how to write generic structures using functors, supporting the notion of software reuse.

5 Models and Paradigms

Programming languages allow programmers to construct models of reality in order to frame a solution in terms of the problem domain. For instance, languages usually allow us to think in terms of integers and floating point numbers, hiding the underlying representations. SML's powerful data modelling facilities are consistent with an emphasis on modelling the environment that needs to take place as part of understanding customer requirements.

The arguments in favour of using a declarative paradigm have been well rehearsed and involve the ability to abstract away from much of the normal "housekeeping" to do with memory management. This allows us to teach first years to

focus on problem solving strategies. There are problem domains, however, for which a declarative solution is not the most natural and straightforward. A large class of such problems involves interaction with the user [3]. Such problems can of course be avoided in the examples we give to our students but it is our aim that students should ultimately be able to switch between paradigms as appropriate. The most productive, expert programmers do this and experience in using different paradigms and languages is more important than which actual language is taught first [5].

5.1 Concurrent material

At the same time as learning to program in SML, the students are also following one course on digital electronics and one on mathematics. Additional lecturing time on the programming course is spent on general computing topics such as using common applications packages, understanding the hardware organisation of a simple computer, understanding system software and discussing the role of IT in society. Digital electronics gives an understanding of computer hardware primarily at the level of digital logic, but also covers some assembly language programming. The mathematics course looks particularly at discrete mathematics and its relevance in computing.

6 Metaphors

Important for learning are the metaphors implicit in the content of the course, and whether or not they form a coherent system with those metaphors of computing already learned or being learned concurrently. We have seen that the background of the students varies. Most have an A level in mathematics, or equivalent, and qualifications in scientific or technical subjects. Another variable is whether or not they have a prior substantial knowledge of computing.

6.1 The Calculator

Perhaps the most important metaphor the students meet is that of the *calculator*. SML allows you to type in expressions, such as

```
3 + 4;
```

and responds with a message such as

```
val it = 7 : int
```

As well as evaluating the expression and printing the answer, the system infers the type of the expression (`int` standing for integer) and tells the user. It is possible to use an editor to prepare a file of SML statements and to load the file

in. Initially we do not teach the students to do this, but show them how to use the system directly.

We made this decision because we felt that this gave immediate access to the system without the need to explain how to use an editor, or what a file is. All students are familiar with using calculators and have a mathematical or scientific background and so we assumed they would be happy to be introduced to a language through numbers. Reinforcement of knowledge about types was important and helpful.

We did not take sufficiently into account the fact that many students already have a model of what is involved in programming. For those who have used Pascal in a turbo-like environment, this involves using pull-down menus to compile source code and the object code may be seen as an entity in its own right, executed by a separate command. Our students were learning to use SML on a UNIX system running Motif. Part of their environment, then, presented metaphors which cohered with previously learned ones (windows, menus, mouse), but part of it did not. There was, therefore, an overlapping of different systems of metaphors - different, but alike enough to have the potential for confusion as users shift between the two. The apparent simplicity also had the undesireable effect that some students did not take in some of the important concepts that were being taught in the first two weeks. When they reached the material on higher order functions, they perceived an apparent discontinuity in the level of difficulty.

It may be objected that the metaphor of a calculator is also wrong on the grounds that it focuses on numbers, which is not what Software Engineering is all about. In the first stages of a large software engineering project an understanding of the environment in which the system must operate is essential in order to understand the customer requirements. The approach may be defended on the grounds that working with numbers does relate to the immediately previous experience of the student. It also allows us to present examples using the simple data types provided by the language and almost immediately we give examples which involve values of type `string`.

A stronger argument for keeping the calculator is that it helps one to present the idea that a program in SML is an *expression* rather than a series of *commands*. The calculator thus becomes part of a coherent system of metaphorical concepts. The declarative way of thinking can be difficult at first for those used to procedural programming, but students did successfully switch into the new paradigm. Once they grasp the notion that a program is an expression, we can show them how to replace one expression with another and present a *substitution model*. Most students find this helpful for understanding how a complex expression might be evaluated, particularly recursive function calls with list arguments. Two approaches are open to us to overcome any difficulties. We could insist that students only use files of SML commands prepared using an editor, or even change language and use, say, Miranda, where a *script* has to be created. Another approach is to explain more clearly to the cohort why we feel the use of the

calculator metaphor is an appropriate one, and what its limitations are. Carroll and Thomas [6] say "When introducing a metaphor, explicitly point out to the user that it is not a perfect representation of the underlying system and point to the limits of the metaphor." They suggest that highlighting the provisional nature of a metaphor makes it easier for someone to acquire a more sophisticated view at a later stage.

6.2 The Environment

The set of value bindings in force when an expression is evaluated constitutes the *environment* of an expression. A *let* expression evaluates an expression in a locally modified environment. The idea of an environment is metaphorical, where we picture a *place* containing various identifiers. The aim is to keep this environment as *uncluttered* and *neat* as possible. This is consistent with any previously learned notions concerning program states, and students used local declarations well and without problems.

6.3 Layers

Another metaphor is that of a system built up from different layers of abstraction. In the first lecture we say that there are several layers of software in a typical computer system and we explain the provision of types as an abstraction away from the actual hardware. This is a standard computing metaphor [7]. It is reinforced by the general Computer Systems material taught in the rest of the programming course, but further support would be given if material on, say, Operating Systems, were to be taught concurrently.

7 Conclusions

We conclude that we should have spelled out some of the metaphors explicitly in order to make them more effective or to dispell confusion about why they are being used. We would also recommend juxtaposing more material supporting the notion of layers of abstraction. A new modular structure to be introduced in 1994/5 will allow us to do this and to bring forward a course on operating systems principles to the first semester. In spite of these recommendations, we deem the introduction of SML to have succesfully fulfilled its purpose in providing a firm basis for the teaching of Software Engineering principles.

References

[1] 19.

[2] George Lakoff and Mark Johnson. *Metaphors we live by*. University of Chicago Press, 1980.

[3] Meurig Beynon. Paradigms for programming. In *Proceedings of Workshop on Functional and Logic Programming Languages*, 1986.

[4] Ron Brunt and Andrew Hutt (eds). *OPENframework The Systems Architecture. An Introduction*. Prentice Hall, 1992.

[5] Marian Petre. A paradigm please - and heavy on the culture. In *Proceedings of NATO Advanced Research Workshop on User-centred Requirements for Software Engineering Environments, 1991*. Springer, 1994.

[6] John M. Carroll and John C. Thomas. Metaphor and the cognitive representation of computing systems. *IEEE Trans. on Systems, Man and Cybernetics*, 12:107–115, 1982.

[7] Andrew Tanenbaum. *Operating Systems. Design and implementation.* Prentice-Hall, 1987.