MICHAEL THOMAS FLANAGAN & JAN SMITH

7. FROM PLAYING TO UNDERSTANDING

The Transformative Potential of Discourse Versus Syntax in Learning to Program

INTRODUCTION: THE GAMES STUDENTS NEED TO PLAY

First year electronic engineering students are no exception to their contemporaries in their addiction to computer games. However, their mastery of these lucrative products of object-oriented programming (OOP) does not readily translate into an understanding of OOP when presented as a formal first year course. One of the fascinations of teaching programming is that, whilst many students learn to program without apparent difficulty, a significant proportion finds the activity extremely troublesome. This observation may be compounded for students of electronic engineering, where threshold concepts (Meyer and Land, 2003, 2005) may be 'nested' in the curriculum. Such potential thresholds may lie in the concepts of OOP, in the exemplifiers dictated by electronic engineering syllabi or in the linguistics of a computer language itself. Implications for teaching and curriculum redesign vary significantly across this spectrum. In this context, students' problems appear to arise from two sources: firstly, the form of the programming language which, to paraphrase Andersen (1990), parasitises English but cannot be read as English, an overwhelming threshold conception, or secondly, more localised threshold concepts inherent in OOP itself, such as abstract classes and interfaces. Consequently we have adopted a three-fold schema to discuss these potentially troublesome concepts (Figure 1).

The focus in this chapter is on our third stream: students who find that the language itself is a threshold, and cannot make sense of the game's rule book. These are our operationally challenged students (Smith, 2006). This stream will be discussed in the context of a linguistic challenge. The more localised thresholds associated with the first two streams will be discussed elsewhere.

CONCEPTUAL AND OPERATIONAL CHALLENGES

An analysis of both common mistakes and of the examination questions successfully completed by conceptually and operationally challenged students suggests the problem is deeper than a simple failure to appreciate the role of the individual components of the algorithm, e.g. data structures, data manipulation instructions, conditional expressions, control structures, etc. The language itself appears troublesome, and what they face is a translation problem. The students are highly qualified and many in the conceptually and operationally challenged

R. Land, J.H.F. Meyer & J. Smith (Eds.), Threshold Concepts within the Disciplines, 91–103.

streams are among our most successful students in other courses. The motivation of the bulk of such students appears strong, as indicated by their high attendance rate and continuing determined efforts to complete each programming exercise over two years.

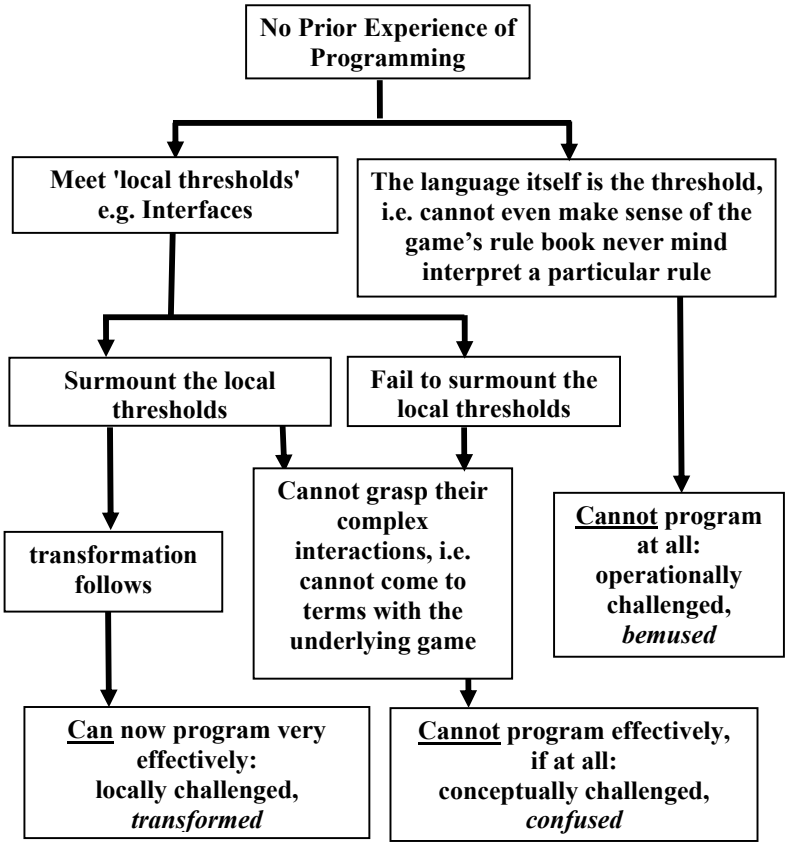

Figure 1: Threefold schema of computer language learning problems

In hypothesising that this situation is a language problem we have looked at our chosen language, Java. In common with most computer languages, it is based on a context-free grammar and, in its formal specification (Gosling et al., 2004), its authors describe its lexical and syntactical structures in such terms. This facilitates its specification, its implementation and, more generally, the discussion and resolution of such problems as undecidability. It has also been said that the common base of context-free grammars 'has also been critical in making it readily possible for people to learn such languages' (Wolfram, 2002). However, close observation of the common mistakes of the operationally challenged stream

suggests that we may more fruitfully borrow from a different tradition in semiotics and linguistics in attempting to analyse the problem. In coding such components as conditional expressions, iterative loops and variable declarations, student errors bear a resemblance to linguistic problems in natural languages as discussed in terms of the semiotic concept of the markedness of oppositional pairs (true/false, public/private) and of synonym definition (if/whether) (Chandler, 2002; Tobin, 1990; Battistella, 1996).

THE THEORY OF MARKEDNESS

> every single constituent of any linguistic system is built on an opposition of two logical contradictories: the presence of an attribute ("markedness") in contraposition to its absence ("unmarkedness") (Jakobson, cited in Chandler, 2002, p. 110)

Markedness is now regarded as linguistically central (Trask, 1999). Table 1 summarises the differences between the marked and unmarked forms in such natural language relationships. The marked signifier is generally the more complex and the one most likely to present problems in learning.

Table 1. The marking relationship (after Stross, 2005)

Unmarked	Marked
Greater frequency of use within language (dominant form)	Lesser frequency of use
May not be overtly marked	Will be overtly marked
The implied in an implicational relationship	The implier in an implicational relationship
Less complex morphologically	More complex
Appears in neutralized context	Does not appear in neutralized context
Early child acquisition	Late acquisition, more difficult
Usually first added and last lost in language change	Last added and first lost

The dominance of the unmarked over the marked is often expressed as the number of occurrences of the unmarked form as a percentage of all occurrences of both within a significant example of the language (Chandler, 2002, pp. 110-115). Our corpora for comparison are the world wide web and Flanagan's Java library (see Table 2).

The true/false pair is only weakly asymmetric in Java and the students neither perceive the concept of a Boolean pair as difficult nor do they actually find it so. However when we come to the keywords public and private, public is highly dominant and although operationally challenged students have few problems with public, they certainly find the marked sign, private, problematic. They do not,

however, self-identify this as a problem. In Java, public signifies that the following variable has no restrictions on its access, i.e. no special methods have to be written to assign a value to it or to obtain its value - the neutral form. The keyword, private, signifies that the following variable does require such methods to be incorporated into the class in which the private variables are declared. The public/private problem becomes much more troublesome for operationally challenged students when extended to include the even more asymmetric public/protected pair. Protected is a keyword signifying that such special methods are needed as for the private keyword, unless the variable it referred to benefits from object-oriented concept of inheritance, i.e. an even more cognitively complex signifier.

Table 2. A comparison of dominance within markedness between English and Java. The percentage figure is the dominance of the unmarked form as a percentage of both forms

Sign pairs in English	**Occurrence on www June 2006**	**Sign pairs in Java**	**Occurrence in Java library**	**Student perceived level of difficulty**	**Observed student level of difficulty**
true / false	1.2 billion / 0.4 billion [75 %]	true / false	1082 / 918 [54 %]	low	low
public / private	5.2 billion / 1.9 billion [73 %]	public / private	2959 / 756 [80%]	moderate	high
public / protected	5.2 billion / 0.6 billion [90 %]	public / protected	2959 / 223 [93%]	moderate	very high
		absent implied signifier / static	~4000 / 1000 [~80 %]	low	very high
		float/ Float	283 / 24 [91 %]	moderate	very high
		double/ Double	4163 / 365 [92%]	moderate	high

Computer languages that do not have a need for a private signifier do not have a designated public one reinforcing the semiotic analogy that we are adopting as this resonates with Derrida's oppositional logic of binarism in which neither term makes sense without the other (Derrida, 1976).

Difficulties with these issues stand out in operationally challenged students' exercises and exams. Variables, on their first use, must not only be preceded by a keyword signifying its accessibility but must be preceded by a keyword indicating its type, i.e. integer, floating point number, alphabetic character. To locally

challenged students, such declarations are logical if somewhat annoying, but to conceptually and operationally challenged students who we suspect are looking at Java as they would look at English, or mathematics at best, such declarations are problematic. They would not declare each word as noun, verb, adjective or whatever, on first using it in an essay and they would not declare numbers as integer or floating point in a mathematical calculation.

Thimbleby's (1999) Critique of Java suggests that:

> there are two quite different sorts of serious problems facing the Java programmer, **barriers**, which are explicit limitations to desired expressiveness, and **traps**, which are unknown and unexpected problems. Typically, a barrier reveals itself as a compile time error, or in the programmer being unable to find any way to conveniently express themselves. A trap, however, is much more dangerous: typically, a program fails for an unknown reason, and the reason is not visible in the program itself.

He exemplifies barriers with the rules dictating where statements importing packages (pre-existing collections of Java classes) must occur. Thimbleby discusses this in terms 'of the design choice having a negative effect on the explanation of Java (and, by implication, on the learnability of the language)'. In our experience this does translate, in operationally challenged students' programs, to the incorrect placing of this statement: they place it where first needed rather than at a prior position dictated by the Java rules. Thimbleby's barriers tend to match the mistakes made by our operationally challenged students, his traps tend to match locally challenged student errors.

The next simple oppositional pair in Table 2 'absent implied signifier'/static, is highly asymmetric, causes immense problems for operationally challenged students but again elicits few comments in questionnaires. The key word static precedes a method that is general, e.g. a mathematical function such as sine. Its opposite, the absent implied keyword, indicates an 'instance' method, that is a method that can only be associated with an instance of a specific software object, e.g. the method that opens a window (the instance of an object) in a computer's browser. In object oriented programming, it makes sense that the unmarked signifier is the one most naturally associated with an object, so natural as to not even warrant a keyword. However, the natural unmarked English opposite of static, at least for engineers, is not 'instance', 'per-object' or 'non-static', all terms used to denote the absent signifier in program documentation and textbooks, it is the word dynamic (70% dominant). This is reflected in the incorrect discussion answers of most operationally challenged and many conceptually challenged students where dynamic is commonly and explicitly offered as the opposite to static and implicitly, if somewhat incoherently, in their attempts at programming.

Keyword and variable names may start with or without an initial capitalised letter but the two forms are not interchangeable. The keyword float (91% dominant) signifies that the following variable is a simple floating point number which is referred to as a primitive data type. The word Float signifies the following

variable is an instance of an object that contains a floating point number but possesses other properties as well. The float/Float pair exemplifies the alien difference between a natural language and a symbolic language and the interactive complexity of the latter. In English the failure to capitalise the initial letter of a proper noun may raise an eyebrow but have few other consequences. In Java, failure to recognise the difference between a float variable and a Float variable can be very serious. It could, for instance, lead to the drawing of fallacious conclusions in equality testing. If we test whether two identical primitive data types are equal the response will be yes (true). However if we test whether two Float objects with identical properties, i.e. containing the same floating point number, are equal, the answer is no (false). The equality check, in the case of objects, checks that their identities – in effect, their locations within the computer memory – are the same, but does not seek to identify their contents.

THE PROBLEM OF PROGRAM CONTROL AND SYNONYMOUS OPERATIONS

The conceptually and operationally challenged students also commonly find problems with two sets of pair statements that may appear as synonyms but cannot always be treated as such. These are the if and switch statements and the for and while statements. What, for the locally challenged students, represent problem-free and useful statements allowing control over the flow of their programs become, for conceptually and especially operationally challenged students, statements that, though not initially perceived as problematic, typically result in blocks of coding which start as one form, e.g. an if statement, but drift through nonsense code to end as the other half of the pair, e.g. a switch statement. The resonance here is with the work of Tobin (1990) in which he discusses the lexical and grammatical problem of the difference between if and whether in English. He believes that the highly asymmetric choice of the use of if or whether is not arbitrary but motivated by a subtle semantic distinction which revolves around the way in which they are perceived within continuous or discontinuous space. The marked member, whether, presents possibilities that occupy a continuous abstract internal space whereas the unmarked member, if, offers no such semantic integrality. The unmarked form offers possibilities perceived as general and the marked form offers the cognitively more difficult process of perceiving the possibilities as part of an integral set.

Table 3 compares these English and Java pairs showing similar dominance in analogous Java ‘synonyms’ and we note that alongside existing if/switch problems, our operationally challenged students show a similar habit in conflating the coding of the for and while statements.

The above markedness analysis strengthens the proposition that some students are reading code as they would natural language. In attempting to gain insight into this misreading it is worth pursuing the ‘semiotic analogy’ a little further. The markedness analysis suggests that students are interpreting computer syntax as a sequence of triadic Peircean signs whereas the earlier, and from the point of view of natural language, less adequate Saussurian dyadic signs would be more

appropriate. Umberto Eco's conception of the closed and open text presents an interesting point at which a reader of an English text and a reader of programming code may be contrasted. Eco's "model readers" on reading an "open text" make up their own mind at many key points in the text, reassessing previous moves from this vantage point (Eco, 1979; Cobley & Jansz, 1997). His "average readers" on reading a "closed text" are offered occasions on which they can make up their own mind but the range of possible interpretations is limited and ruled by a quite rigid logic. Sadly, for those students who appear to be emulating Eco's "average reader" a computer program, to extend Eco, is a 'totally closed text' in which the many opportunities at each point in the narrative are not present, the rules are absolute and all-embracing. "

Table 3. A comparison of dominance within markedness between two English conditional conjunctions and two pairs of Java program flow control statements

Sign pair in English	**Occurrence on www June 2006**	**Sign pair in Java**	**Occurrence in Java library**	**Student perceived level of difficulty**	**Observed student level of difficulty**
if / whether	7.1 billion / 0.9 billion [89 %]	if else / switch	884 / 64 [93 %]	moderate	high
		for / while	1714 / 194 [90 %]	moderate	high

The frustration that arises when students unwittingly read program code as natural language commonly leads to requests along the lines of: 'can we have some notes, like some other courses, which we can learn the subject by rote'. In other words, the students want us to facilitate their mimicry. On being refused, the request changes to one for some kind of template that would convert rote learning 'mimicry' into 'programming'. It is a measure of their real frustration and brings to mind Andersen's (1990) comment on comparing computer and natural languages whilst discussing the human-computer interface:

> Complete descriptions of natural semiotic systems rarely exist, and in any event their expressions cannot generate the object signs through a causal chain, although folklore has often dreamed of that kind of sign: they are called spells.

Some operationally challenged students, when really dispirited, would like a 'spell' but would settle for mimicry as second best.

In this chapter we have exemplified the usefulness of a markedness analysis in defining those aspects of programming, that when taken in their totality, present the operationally challenged student with an overwhelming threshold conception which may equate to what Perkins (2006, p. 42) terms 'an underlying episteme':

> As used here, epistemes are manners of justifying, explaining, solving problems, conducting enquiries, and designing and validating various kinds of products or outcomes.

Frustrated and unable to access this fundamental episteme, the operationally challenged students then, in trying to solve their programming errors, compound their troubles by further mis-identifying their source.

METAPHORS AND MISTAKES

Dijkstra (1989) recognised the problem of the overall complexity of programming as a barrier to learning but sadly expressed it in terms of a rejection of metaphor and imagination that most now, we believe, would see as unhelpful (Travers, 1996). However, in this, he did raise the problem of the use of language with an instructive example:

> We could, for instance, begin with cleaning up our language by no longer calling a bug a bug but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz. with the programmer who made the error.

Our students have major problems interpreting error messages but we do not believe that this is simply a problem of an inappropriate metaphor. It may be a small component of the problem − we all prefer to think that the other person/machine is at fault. However changing *bug* to *error* will not significantly alter the relationship between the student and the computer. Students do not need anthropomorphic prompts to believe errors lie with a faulty machine but their response to error messages does bring firmly into consideration the student-computer interface. The response to error messages, especially that of the conceptually and operationally challenged students, is initially not to blame the machine but often one of perplexity, even paralysis. They will stare uncomprehendingly at what to experienced programmers is a clear well documented error message or set of messages. This response is one of total failure of signification. A typical error message is shown in Figure 2 with added explanatory boxes (the student would see the message without the boxes).

Maybe the failure of signification is a combination of, to non-experts in Java, unfamiliar signs (exception instead of error, symbol to signify the name of a variable, method, or class, the use of the object oriented dot convention instead of a standard English phrase with spaces between words, thread - concept met but not developed in most introductory engineering courses), a juxtaposition of signs, some present coincidently (the computer name, both the operating window line number and the Java code line number) and, possibly, an overall visual appearance that suggests an abstract iconic sign when none exists.

Student response once the ‘paralysis barrier’ is surmounted is even more instructive. The multiplicity of errors that are often generated reinforces their transference of the origin of their errors to the machine - “I cannot possibly have

made so many mistakes". This highlights a difference between a native language and a computer program and occasionally can lead to a transformation in the students' understanding of programming. Make a mistake in speaking such as using a wrong tense, the incorrect sentence will sound odd but the sense of the conversation will probably not suffer greatly. A corresponding error in Java, e.g. an incorrect variable declaration, may not only prevent compilation but generate a cascade of error messages as the subsequent code may no longer be interpretable by the compiler even though it contains no syntactic errors. Students are advised that they should always correct the error indicated in the first occurring message and then recompile the program - with luck the remaining error messages will evaporate. The operationally challenged students, by and large, do not do so - time and time again they obsessively work through the complete error message list. However, a few students have commented that the 'error message evaporation' sparked an appreciation of the interactive complexity of a program and that consequently such exercises of first constructing a flow chart directing them through this complexity were no longer seen as unnecessary.

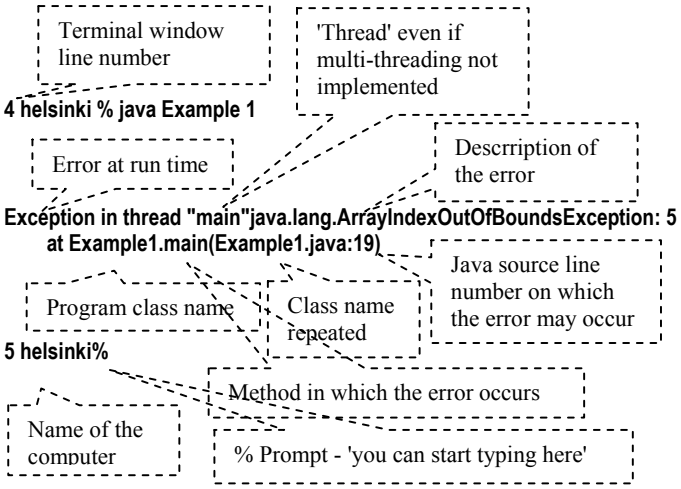

Figure 2. Typical Java error message

More recently Christian Holmboe (2002, 2005) has also concluded that teaching object-oriented design should be treated as a linguistic exercise. He starts with the early Wittgenstein and his language game. This is a fruitful area for those interested in object-oriented design as there are many analogies between the formal definitions of object-oriented design (OMG, 2001) and the postulates on objects within the Tractatus (Wittgenstein, 1921). Holmboe importantly contrasts the specification and learning of object orientation with the 'early Wittgenstein' and

the 'late Wittgenstein' stressing Wittgenstein's realisation that language and meaning were constructed in social practices rather than from logical reasoning. His conclusion that 'the logically perfect language of class diagrams are not as close to natural thinking as may have been intended' is highly relevant to teaching languages such as Java. His statement that 'it can be shown that students of data modelling struggle with the pragmatics of their prior linguistic experience when trying to fit their experiential world into categories and classes' gels well with the conclusions we are drawing, at least for our operationally challenged students. However, we believe them not to be peculiar to object-oriented languages, but to be generally true of high level languages as revealed by a more basic semiotic analysis, although object-oriented design does greatly compound the problems.

THE SEMIOTICS OF PROGRAMMING AS A THRESHOLD

Transformative Aspects

In contrast to the locally challenged and conceptually challenged streams, whose localised thresholds arise out of the compounding of specific OOP troublesome concepts, e.g. interfaces, with identified troublesome concepts in the applied physics underpinning the computing exercises, e.g. complex numbers or electric fields, for the operationally challenged students the language itself is the overwhelming threshold. Those that do cross this threshold, though they may still have to meet the more localised thresholds, in coming to terms with the interactive complexity of programming languages undergo a transformation that facilitates an understanding not only of the local structures, such as the control statements discussed above, but can now grasp many of the structures that underpin computer languages at a deeper level, e.g. methods, object-oriented classes.

Integrative Aspects

The transformative aspect that leads on to an ability to grasp the underpinning structures such as methods and classes facilitates the integration of such concepts into more all embracing key procedures such as the instantiation (creation in software) of an object.

Discursive Aspects

Electronic engineering students at UCL are expected to attain a high level of programming knowledge and skills. However computer science is not their discipline and consequently they do not benefit, as do computer science students, from being in an environment that facilitates their embracing of the ethos of the curriculum, a possible factor in the differing ways computer science and non-computer science students negotiate the liminal space of this overwhelming threshold. Nonetheless, there are common features and we have, as have some computer science departments (Hanks, 2006), moved from requiring students to

work singly to work as pairs or threesomes. We have additionally moved to a more project based approach. We have repeatedly seen students who having grasped a local threshold concept themselves enthusiastically and volubly attempt to lift their partners over the same threshold.

Bounded Aspects

The identification of the bounded nature of the more localised compounded thresholds does present some problems but this is not true of the overwhelming linguistic threshold. The notion of boundedness may best be illustrated by the use of specialist terminology that acquires a meaning in one subject that clashes with everyday usage. One such as example, commonly perplexing our students, relating to computing, is the term 'deprecate'. Whilst common usage imbues this word with negative connotations, in computing it simply means to let an aspect of programming gently wither away, e.g. the retention of an outdated method by its replacement for many revisions and updates of a programming language. The concept of deprecation, then, is bounded by its context of use. This example reinforces our semiotic approach and further examples could be drawn from several of the markedness examples discussed above.

Reconstitutive Aspects

The locally challenged students generally identify the localised compounded thresholds that they meet irrespective of whether they surmount such thresholds. The operationally challenged students do not even correctly identify their troublesome concepts as outlined in Tables 2 and 3 until they surmount the overwhelming threshold. Then, both they and their class facilitators can recognise the transformation demonstrating a shift in learner subjectivity. Our experience of observing such a transformation is that it is *irreversible* and the *troublesome*, *incoherent* and *alien* aspects (Perkins, 2006) have been discussed in the markedness examples presented above.

IMPLICATIONS FOR TEACHING PROGRAMMING IN ENGINEERING CONTEXTS

Our analysis of operationally challenged students suggests that we should introduce the language game formally to such students. Indeed Holmboe has begun to introduce the language game to his object-oriented design students including readings from Wittgenstein but he is addressing a more mature and main stream computing class. However we believe that a gentler introduction would work with engineering students and evidence of the validity of such an approach may be gained from a second year course which includes an introduction to fuzzy logic controllers. Fuzzy logic would, at first sight, appear a likely candidate as a threshold concept for engineering students. It aims to turn imprecise – fuzzy – statements into the formalism of set theory and program code. This is such a strange progression for engineering students that the formalism of the method is

preceded by a discussion of language as we use in it everyday life and how we might reduce it to a set of rules that may then be implemented digitally. This precipitates a very interesting discussion and fuzzy logic not only causes very few problems in the course but is the most popular topic as judged by the statistics of the optional questions answered in the examinations.

The direction of much of the discussion that informs the teaching of programming in computer science – such as which language to teach first, whether a knowledge of computer architecture is an essential prerequisite, etc. – is relevant to our concerns, but we believe the linguistic and 'nested' nature of troublesome concepts for engineering students cuts across these dichotomies. Consequently, further research into the tipping points of our three streams is required to enable the design of exercises addressing threshold concepts and these may differ quite significantly for the three streams. The above analysis has been matched by analyses of the other two streams leading to a clear appreciation of the nature of the compounded local thresholds but we have yet to achieve a meaningful synthesis across this spectrum. We have, as yet, a poor grasp of how the semiotic problems impact on the incorporation of grasped local thresholds into the overall threshold conception. Work is in progress on this synthesis.

REFERENCES

Andersen, P. B. (1990). *A theory of computer semiotics, semiotic approaches to construction and assessment of computer systems.* Cambridge: Cambridge University Press.

Battistella, E. L. (1996). *The logic of markedness.* Oxford: Oxford University Press.

Chandler, D. (2002). *Semiotics: The basics*. Abingdon: Routledge.

Cobley, P. & Jansz, L. (1997). *Introduction to semiotics*. Royston, UK: Icon Books.

Derrida, J. (1976). *Of grammatolgy* (transl. G. C. Spivak). Baltimore, MD: John Hopkins University Press.

Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. *CACM, 32*(12), 1398-1404.

Eco, U. (1979). *The role of the reader: Explorations in the semiotics of texts*. Advances in Semiotics. Indiana University Press.

Flanagan, M. T., *Michael Thomas Flanagan's Java scientific library*. Retrieved on 10 June 2006 from the World Wide Web: http://www.ee.ucl.ac.uk/~mflanaga/java

Gosling J., Joy B., Steele G., & Bracha G. (2004). *The Java™ language specification*, third edition, Boston: Addison-Wesley. On-line version: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

Hanks, B. (2006). Student attitudes toward pair programming. In *Proceedings of the 11th annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Bologna, 2006.

Holmboe, C. (2002). Revitalising old thoughts: Class diagrams in light of the early Wittgenstein. In J. Kuljis, L. Baldwin, & R. Scoble (Eds.), *Proc. PPIG 14*, 14th Workshop of the Psychology of Programming Interest Group, Brunel University, June 2002, pp. 196-203.

Holmboe, C. (2005). The linguistics of object-oriented design: Implications for teaching, Annual Joint Conference Integrating Technology into Computer Science Education, in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Caparica, Portugal, pp. 188-192.

Meyer, J. H. F. & Land, R. (2003). Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising within the disciplines. In Rust, C. (Ed.), *Proceedings of the 2002 10th*

International Symposium on Improving Student Learning Theory and Practice − 10 years on (pp. 412-424). Oxford: Oxford Centre for Staff and Learning Development.

Meyer, J. H. F. & Land, R. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education, 49*(3), 373-388.

OMG. (2001). *Unified modelling language specification v1.4.* Needham, MA: OMG ObjectManagement Group. [OMG web page: http://www.omg.org/]

Perkins, D. (2006). Constructivism and troublesome knowledge. In Meyer, J. H. F. & Land, R. (Eds.), *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge.* London and New York: Routledge

Smith, J. (2006). Lost in translation: staff and students negotiating liminal spaces. SEDA Spring Conference 2006: Advancing Evidence-Informed Practice in HE Learning, Teaching and Educational Development, 8-9 June 2006.

Stross, B. (2005). Introduction to graduate linguistic anthropology course. Retrieved on 10 June 2006 from the World Wide Web: http://www.utexas.edu/courses/stross/ant392n_files/marking.htm

Thimbleby, H. W. (1999). A critique of Java. *Softw., Pract. Exper. 29*(5), 457-478.
On-line version: Thimbleby, H. W. (1998). A critique of Java, Retrieved on 10 June 2006 from the World Wide Web: http://www.uclic.ucl.ac.uk/harold/srf/javaspae.html.

Tobin, Y. (1990). *Semiotics and linguistics.* London and New York: Longman.

Trask, R. L. (1999). *Key concepts in language and linguistics*. London: Routledge.

Travers, M. D. (1996). Programming with agents: New metaphors for thinking about computation. PhD Thesis, Massachusetts Institute of Technology. On-line version: http://alumni.media.mit.edu/~mt/diss/index.html.

Wittgenstein, L. (1921). *Tractatus logico-philosophicus* (transl. C. K. Ogden). London: Routledge.

Wolfram, S. (2002). *A new kind of science.* Wolfram Media.

AFFILIATIONS

Michael Thomas Flanagan
Department of Electronic & Electrical Engineering
University College London

Jan Smith
Centre for Academic Practice & Learning Enhancement
University of Strathclyde