

# The Abstract is ‘an Enemy’: Alternative Perspectives to Computational Thinking

Alan F. Blackwell

*Computer Laboratory  
Cambridge University  
Alan.Blackwell@cl.cam.ac.uk*

Luke Church

*Computer Laboratory  
Cambridge University  
luke@church.name*

Thomas Green

*Dept of Computer Science  
University of Leeds  
greenery@ntlworld.com*

Keywords: POP-IA. Social organisation and work; POP-VA. Attention investment

## Abstract

An enemy.

## 1. Introduction

What is the worst possible name for a software module? (As the first author asks his first year software design class). Answer: “ProcessData” – which is to say, “this is a module that processes some data”. This surprises students. In their view, ProcessData might even be the *best* possible name for a module, because it is the module that can do anything, is infinitely adaptable, and for which later implementation detail can account for all necessary variation. The interface to such a module is completely generic, and hides all detail, just as we ask them to do. In Alan’s professional career, it was far from unusual to see this and similar names in the source code of deployed systems, as module names, function names or method names. He almost certainly used similar names himself, as a young programmer, although later learned to avoid them.

Why are generic names so bad? As Alan tells his class, a generic name demonstrates an abdication of responsibility by the designer. Giving something its proper name is the way that we capture the nature of the thing. In many parts of software design, assigning names (to data and processes) is the hardest task – and one that deserves far more careful attention. The advice of master software craftsmen, such as Microsoft’s Steve McConnell, confirm this. His classic book on practical software construction (McConnell 1993) has many sections with titles such as “Good Routine Names”, “Considerations in Choosing Good Names” and so on. A generic name is not a name – as if Alan had called his daughter “person”, or his cat “cat” (or in fact, “null”, since he doesn’t have a cat). In software design, a generic name later bites the designer, when he or she realises that the purpose of this component is unknown, or that it has been implemented to deliver functionality incompatible with the requirement (a requirement not captured by the generic name).

If this is a problem for software designers, how much worse must it be for their clients? Imagine a system requirements document, created for approval by clients, that includes statements like “the user makes a query”, “the system processes the data” or “a report is generated”. One response is for the client to give up, in the face of relentless jargon, and simply trust the developer that the system will be useful. Another is to “go native”, adopting the computational perspective, and agreeing that yes, these descriptions do appear to capture all possible requirements the client might have. Requirements statements in this form can insulate programmers against the chills and storms of a user-centred design process. If users object to an interface mockup, then the generically-minded programmer can abdicate responsibility by explaining that the UI functionality is generic, and can always be configured later. Or that the colours are not a relevant topic for discussion, because they can be specified by the user at runtime. Or that the syntax of a programming language is only “sugar”, and not a relevant concern in programming language design (which should be concerned with abstract semantics, as noted in a previous PPIG paper (Blackwell & Hague 2001)).

## 2. Overview of Empirical Basis

The empirical basis for this paper is long-term participant observation of four different software design projects, all taking place in Cambridge University. Each of these was a relatively large-scale development, whose scope extended across all departments of the university, with thousands of potential users. Each was developed by a specific central office having responsibility for a particular specialist type of administrative operation across the university. The common concern that makes all four systems interesting objects of research was a tension between the concerns and processes of user centred design, and those concerns and processes of more traditional software project management and organisational change management.

The history of centrally managed software development initiatives in Cambridge has not been characterised by universal success. The first of these four projects, CAPSA, gained some notoriety after a public report described systemic failings across the whole project (Finkelstein & Shattock 2001), many with long-term consequences for the usability of the system. In response to that report, the second of these projects, CamSIS (Cambridge Student Information System) was far more strictly managed. MIS managers placed priority on cost control and preservation of the off-the-shelf functionality, rather than user-centred design. The University Board of Scrutiny later drew attention to the fact that usability had been seriously compromised as a result (Downer et. al. 2007). The Cambridge Research Information System (CamRIS) was, at the time of writing, a project in early stages at the CU Research Services Division. Managerial priorities led to the adoption of the same management processes used for CamSIS, despite the recorded deficiencies of that system. Nevertheless, project team members attempted to introduce selected practices of user-centred design. Finally, the CamTools system is an ongoing local deployment of the open source Sakai platform<sup>1</sup> for educational content management. The Centre for Applied Research in Educational Technology (CARET) has recognised that usability is a critical issue for development of CamTools, and has engaged in a programme of user research and consultation to evaluate concept prototypes.

The authors have had varying degrees of contact with the development teams for each of these four systems, and have some experience as users (or user representatives) with each. We have not held formal roles on the development team for any of them, and our status as participant observers has generally been that of a (more or less) friendly and expert local critic. This has involved meetings with developers, review of specification documents, assistance with broader user consultation, participation in public debate, and preparation of teaching materials for which these familiar local systems are used as exemplars of user interface design issues.

Although the systems themselves have presented different degrees of managerial success, and differing levels of user acceptance, we have noted a number of common features, relevant to our concern with abstraction, that are played out repeatedly. Although the four projects we observed all took place within Cambridge University, we believe that the general dynamic is a universal one, to be expected whenever user-centred design processes are applied in an organisational software context. To that extent, these are general questions for psychology of programming, of a kind that is highlighted by the structure of large institutions, whether or not they are academic institutions.

The issue with which we are most concerned is that best practices in user centred design tend to be grounded in the needs and actions of specific users. The design of software architectures and organisational processes, in contrast, aims to identify, implement and maintain effective data and process abstractions. The ability to abstract over diverse data descriptions and process requirements is an essential skill of the professional systems analyst, and is a primary focus of many system development methodologies and software design techniques.

As a result, our advocacy and assistance with user centred design processes often involved an observed tension between abstract descriptions of the system, and descriptions of specific cases. In some cases, users offered abstract descriptions based on generalisation over their own repeated experience, but these conflicted with the abstractions used by the system developers.

---

<sup>1</sup> <http://sakaiproject.org/>

An example in CamSIS is the hierarchical menu structure to be navigated. One part of the menu structure contained functions related to graduate students. In order to reach this sub-tree, a graduate student had to pass through several higher-level menus, each with only one visible option (because graduate students are not authorised to carry out any other system functions). From the perspective of the developers, this was not a design fault. The overall menu structure was correctly specified, and there was a valid path to all functions. The fact that intervening navigation levels were perceived uniformly by graduate students as a navigational obstacle was not consistent with the abstract description of the developers. When faced with a complaint by a graduate student, this was interpreted as a specific case (the complaining student), not a generic case (which could not be accommodated because it would require modification of the abstract structure of the user interface).

Even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centred design. In the case of CamRIS, one of the main priorities for system developers was to incorporate appropriate costing and accounting processes for the management of research. These processes were often incompatible with the processes by which research was actually conducted. As a result, functionality related to a single type of research project might be scattered throughout the system, because it related to different accounting transactions. User consultation exercises were compromised because they, too, were structured according to the abstract descriptions of the department commissioning the software. A series of consultation workshops was proposed, devoted respectively to sponsor identification, proposal costing, and project management stages of research projects. Academics were invited to choose one of these workshops, and attend the one that they were most interested in. Of course, this is a poor fit to the needs of academics, who are engaged in all of those stages of a research project. Academics would rather attend a meeting at which they shared their needs among other academics, and have separate workshops oriented toward the needs of secretarial, administrative or accounting staff. The development managers did not recognise this distinction, because the workshops were structured in a way that was completely consistent with the structure of the Research Services Division itself. That structure represented an abstract conceptualisation of the process of doing research; the specific experiences of system users inconveniently crossed categories and thus defied classification according to the 'correct' accounting abstractions. This can be described in terms of Conway's Law, summarised as "Any piece of software reflects the organizational structure that produced it" (Conway 1968).

In many cases, organisational abstractions are made more rigid through their expression in computational form, resulting in an amplification of abstract structures through the reinforcement of Conway's Law. Database schemas and system architectures can easily be described using abstract or generic terminology that fails to account for variation in user experience and requirements. This occurs even where developers are very well-intentioned. The CamTools team took great care to emphasise that users of the system would not have the same concerns as the developers themselves, and that they were not developers but "people", wanting to do their "ordinary job in the university". Unfortunately, even this formulation resulted in accidental avoidance of specific user needs. The abstraction of "people doing jobs" fails to make a clear distinction between the needs of university lecturers and university students, who often have quite different roles as primarily creators and primarily consumers of the content being managed. Because Sakai has an admirably democratic model of university life, in which all users can both create and consume content, the more generic abstractions are hard to reconcile to more mundane student experiences.

We have now seen several examples (CamSIS' menu structure, CamRIS' division of research into accountancy steps and CamTool's generalisation of users into 'people doing their jobs') where the process of abstracting away from the user has hampered the usability of the resulting system. The difficulties in all of these examples could be described as 'misfits' (Blandford & Green, 2001). A misfit is a correspondence problem between abstractions in the device, abstractions in the shared representation (the user interface) and abstractions as the user thinks about them. Here, the abstractions in the 'shared representation' (the user interfaces for CamSIS, CamRIS and CamTools) don't match the users' way of thinking about the world. Such misfits are known to cause usability difficulties.

We shall now consider the process that, we hypothesise, contributes towards these misfits.

### 3. Inconvenient Complexity

The abstractions underlying computational systems are one of the points at which the behaviour of the system is negotiated (Brown 2001). Simpler models are generally favoured by the implementers, with good reason: as Brooks observed (Brooks 1974), the world is complex and often arbitrary. At some point during the design of any reasonable computational model, simplifying assumptions will have to be made.

Unfortunately, clean technological models of the world generally do not offer a good fit to human ones, for example 'student' refers to extremely complex sets of relations in a modern university. From our human perspective it equally applies to a 'probationary PhD student' as to a person who is thinking of applying for an undergraduate degree at a women's college but has taken an assumed name which, unknown to them, would normally imply that they are of the opposite gender. These complexities are inconvenient when trying to design representations that can be interpreted computationally.

Therefore, the first step of the 'process' is for developers to abstract away this inconvenient complexity, in order to reach the computational essence of the domain being represented.

#### 3.1. Step 1: Abstract Away the User

This abstraction away from entities in order to represent them to a computer results in the 'user' becoming a somewhat simplified version of what a real user is. Unsurprisingly, complexities that are inconvenient or difficult to represent get smoothed over. For example, the complex ways in which people use aliases get replaced by the simple 'name as composition of first name(s) and surname'.

Once the inconvenient complexities are ironed out and all users represented in a single form, they can be aggregated. This aggregation occurs both in software, and in the processes that designers use in discussion. The designers' discourse shifts from being a 'design for Fred' to a 'design for the user'. This has a number of effects:

- If too much complexity was abstracted away, then the aggregates may be too inclusive. This is the case with CamTools, where the aggregate user included at least two very different types of people.
- Because of the abstracted and generalised nature of these aggregates, it becomes harder to reason about what might or might not be important to a user; what, for example, is important to the aggregate of a student and a lecturer? This 'user blindness' (Jenson 2002), means that it's hard to prioritise functionality resulting in feature creep, where everything becomes equally important and therefore equally hard.
- The discourse itself becomes confused: it is occasionally unclear whether designers are talking about the user, the physical person, or the user, the computational representation, or both simultaneously.

So with the best practical intentions the specific person has now become the generic user.

#### 3.2. Step 2: Dehumanise the User

The generic user has a number of properties that are different to the physical user. Principally, almost everything about them is unknown. It is, for example, far easier to assume that a generic user thinks about the world the way a developer does, than it is to say that 'Fred from upstairs' does. One of us has seen an extreme situation in which the imagined 'user' was assumed to have the technical skill of a UNIX system administrator, when actually the system was to be used by a human resources administrator.

This process of tending towards the generic, unknown user can have more disturbing effects. If the generic user is merely a composition of a number of more 'primitive elements', such as strings for their name, simple enumerations for their ethnic origin etc., then correspondingly simple constraints can be placed upon them. For example, it is computationally trivial to require that a user select his or

her ethnic origin from a list of three possibilities. Whilst the generic description of this: ‘the user describes their origin’ might be acceptable, the literal description: ‘Fred specifies his ethnic origin as one of White, Black or Asian’ is less likely to be accepted.

Thus, by abstracting away from the user there is the possibility, not only of creating confusion, but also of eliding aspects of people that the people may perceive as being important to their humanity, not just incidental complexities (Anderson 2001).

Just as we saw with the process of abstraction, here the dehumanised nature of the representation of the person also misleads the designers. Examples for this can be found in the failure of security systems when they are deployed into a social context. For example, imagine being in an office which institutes a ‘you must lock your PC session when you leave your desk’ policy. If you’re the only person in the office who follows this new policy, then it sends unwelcome social signals about how much trust your fellow officemates. In this case social considerations are likely to override the ‘appropriate security behaviour’. But if the designer views people as entities that have been abstracted away from their rich social context to an impoverished one, then the consideration of the effects and likely success of such a policy will not occur.

These examples of how abstracting away from, and dehumanising, the user can affect the system have glossed over the final step in the process, imposing the model onto the users.

### 3.3. Step 3: Change the User

The third step in the process happens when the piece of software that has been created interacts with the world at large. There are several conditions in which this can happen:

Firstly, as we have seen in our examples, problematic abstractions of the system may appear in the user interface. These abstractions do not match the users’ way of thinking about the world, and as such cause the mis-fits that we have seen, resulting in usability problems.

Secondly, sometimes to maintain the integrity of the computational model, it is necessary to be assertive in the way the technology is deployed. Security technology is used to try and force the adoption of a behaviour. For example, CamSIS maintains separate users for PhD applicants (who are issued random userIDs and passphrases) and existing undergraduates. An undergraduate who is applying for a PhD might try and login using their undergraduate ID, but would find themselves unable to manage their application this way. They are forced to use their new ‘random ID’ instead.

Whilst this is a substantial annoyance, its effects are relatively minor. More severe problems arise when, for example, the new London Ambulance Service computer aided despatch system attempted to mandate that crews take the ambulances they are assigned to (Finkelstein & Dowell 1996).

Thirdly, and much more subtly, the decisions that were made in the abstracted, dehumanised world may shape the real world by making some things unknowable. Consider, for example, a change in the International Classification of Disease version 9 to version 10 which changed the names of a number of diseases from geographic to systematic ones (Bowker & Star 1999). An unintended consequence was to remove the implicit travel information from a patient’s medical record.

It is not (or should not be) surprising that abstract descriptions have political consequences and can be used to political ends. Bowker and Star’s seminal work in the sociology of knowledge “Sorting things Out”, offers an extensive analysis of the political consequences of classification (Bowker & Star 1999). The first author’s analysis of political metaphors in the Java libraries (Blackwell 2006) has already drawn attention to the fact that social and political concerns may be central to future research in the psychology of programming.

Outside the domain of programming, a few researchers have been working to develop information organisation, presentation and navigation interfaces that avoid inappropriate classification of socially or politically sensitive media. These include Robin Boast’s work on classification of museum artefacts, Turnbull’s (2006) StoryWeaver project derived from critical thinking about maps, and research by Verran and Christie (2007) resulting from the tensions when western classification conventions were applied to Aboriginal cultural knowledge. These projects have generally applied

"flat" visual presentations as an alternative to database schemas or metadata that encode abstractions. At present, none include further "programmable" behaviour beyond the presentation of a navigation interface.

These projects are partially responses to a further danger that arises in the political use of abstractions: the designers of the abstractions think that they somehow capture the 'essence' of the problem. Given that these abstractions are, by definition, simplified representations of the world, this is a dangerous claim to make. Real-world theories of knowledge and language abound in complexity, even though they may share apparent correspondences with far simpler computational conventions. Consider one of the aspects of desirable "computational thinking" which includes 'recognizing both the virtues and the dangers of aliasing, or giving someone or something more than one name.' (Wing 2006). This statement appears to refer to the dangers of aliasing as a programmer would understand them, involving hidden dependencies: if two pointers address the same memory location, then changes via one pointer will affect reads from the other pointer, leading to surprises. But that computational concern has no apparent relation to the political implications of naming in human affairs, for example the implicit tension between career and family when a female academic adopts her husband's surname on marriage, while continuing to publish under her own name.

#### 4. Computational Thinking and its side-effects

In the highly influential recent paper cited earlier, Jeannette Wing (2006) advocated a broad programme to teach 'computational thinking' as the application of concepts and approaches from computer science to help to understand the world. Yet we have seen that the constrained application of abstraction used to mediate the human and technological worlds can cause substantial problems. If we proceeded, without due caution, to re-interpret the world computationally as Wing advocates, what might the effects be? We shall now consider some effects that might occur that are relevant to the concerns of this paper.

##### 4.1. Literalism

The first effect might be a propagation of literalistic thinking. Computers reason literally about the world, as the enriching social context that humans enjoy is largely inaccessible to them. As such, they take everything at 'face value', or in more scientific terms, they are only capable of manipulating the explicitly available syntax and mathematically-structured 'semantics' of information, not its socially constructed counterpart.

Consider for example Steve McCurry's famous portrait of an afghan refugee girl; what might state of the art image processing have to say about this? Little more we suggest than 'portrait', 'girl'. The value of this image, like most images (House et. al. 2005), is in its computationally inaccessible, socially constructed properties, not the numbers that make up the pixel intensities.

But the problem is more serious than this. The design of computer systems makes exhaustive use of metaphors as a source of creative inspiration and a way of attempting to explain behaviour (Blackwell 2006b). However these metaphors are inaccessible to computers. Consequently they receive the same literalistic treatment as everything else. This has the unfortunate consequence of impoverishing not only the intended target of the metaphor, but also the human interpreter.

##### 4.2. Goal Conflation

A further effect of computational thinking is that the same abstraction constructed in the definition of a problem is also used to establish the success criteria. This might almost be viewed as the final stage of abstraction: whether a system is successful is no longer measured by its actual efficacy, but rather by some property of its abstract structure. This results in the computational goal and the stated goal becoming conflated. This kind of thinking already permeates much of computer science:

- As a member of the CamTools project reported: "To make the system usable, what we need is a generic framework"

- To make programming easy, what we need is a precise mathematical description of the language (Sewell 2003)
- As a trained computer scientist himself, the second author is disturbed to recognise this tendency in his own research, as previously presented at PPIG, when he claimed that, to help people write code quickly, what we need is an information theoretically efficient channel (Church 2005).

#### 4.3. The Computer as a Laboratory for Abstract Actors

Sociologist of science Bruno Latour, in his analysis of the scientific laboratory, described the variety of actors that become ‘allies’ of the scientist who needs to defend theoretical claims. Laboratory instruments, experimental participants, and other researchers whose work is cited can all be enrolled in the network of actors that support a given theory. Latour describes mathematical analyses as particularly powerful in this enrolment process, because mathematical summaries can draw on and represent very large numbers of allies. The same applies to the computational abstractions and aggregates that we have described above – abstract descriptions, by rising above the circumstances of any specific instance, offer an illusion of universality and universal support. In this respect, computational thinking can become a misleading foundation for scientific work, through encouraging an abstract ‘laboratory’ that is not founded in any real or human phenomenon.

### 5. Implications for Design

How do we proceed from this critical foundation, to the development of tools that might be of assistance? A number of our observations have emphasised the political causes and consequences of abstraction, where the software designers and developers appear to be complicit in systems that disempower, endanger or disadvantage the end-users of the software. A possible technical response, and one that is often proposed, is that end-users themselves might be “empowered” if given the opportunity to develop their own software. In the rhetoric of end-user programming, the term empowerment is often taken to mean purely technical empowerment (perhaps in the sense of being promoted to the status of “power user”, a term often used for end-user programming and customisation technologies of the past). However, our argument suggests that a major ambition might be the political empowerment that results from the ability to define one’s own abstractions, or at least establish the specificity of self-description, rather than being subjected to the abstractions of others.

But our starting question was whether some abstractions might be harmful in themselves, irrespective of who creates them. It is not clear whether end-users have much to gain from creating abstractions themselves. Blackwell’s Attention Investment model of abstraction use suggests that many users go about their work in an anti-abstract manner, because they have (implicitly) calculated the costs and benefits of an abstract task strategy, and have determined that a direct manipulation alternative better fits their desired profile of risk in technology use. This model bears some resemblance to (and was influenced by) Carroll and Rosson’s “paradox of the active user” – that many users choose not to spend time reading documentation, but instead prefer to proceed with getting their job done (Carroll & Rosson 1987). In the same sense, even users who recognise the potential benefits they could potentially gain from controlling their own abstractions might still choose to proceed without doing so. This might be considered as the “paradox of the direct manipulator”.

Fortunately, the formulation of Carroll and Rosson’s paradox did not lead them to offer a counsel of despair. Carroll proposed instead a new approach to the design of documentation, one that he described as “minimalist instruction” (Carroll 1990). The goal was to assist users in acquiring understanding of a system while simultaneously making progress on their task, by providing documentation that described system principles in the context of real task steps.

A similar approach might be taken to the paradox of the direct manipulator. If end-users are oriented toward the right kinds of direct manipulation operations, these might lead them toward the construction of suitable abstractions. At a very local level, this is analogous to programming by example (Church & Blackwell, submitted), in which the user carries out direct manipulation actions

that are observed by an inference algorithm. The user's actions make immediate progress toward the task, and the system assists in defining ways that they might be automatically repeated by recognition of an appropriate abstract pattern. At the more generic level we have discussed in this paper, we would like end-users to be better able to describe, to the software developers they encounter, the abstract implications of the concrete cases of concern to the end-user. This might be done by allowing end-users to construct their own computational behaviours in concrete terms, and allowing the system to deal with any necessary abstractions. This has some similarity to the "instance-based" programming languages that were once proposed as an alternative to object-oriented programming languages. However, instance-based techniques simply allowed users to start defining abstract classes implicitly, by reference to an instance of that class. In our proposal, we suggest exploring ways in which the user might not define abstractions at all, but simply be guaranteed the opportunity to respond to the abstractions that arise, whether generated by computing professionals or by machines, and whether presented in terms of linguistic abstractions, visual formalisms, or simply sets of concrete instances.

## 6. Conclusion

An influential computer scientist, Jeanette Wing, has recently argued that many areas of human endeavour would benefit from increased emphasis on "computational thinking", and that the development of such thinking should be a new priority in general education (Wing 2006). In this paper, we have observed a number of respects in which a central assumption of computational thinking – that abstract formal descriptions are beneficial – might in fact be antagonistic to reasonable human concerns. One contribution of this analysis is simply to moderate the claims made in support of Wing's position. We do not wish to suggest that the computational thinking agenda is misguided, simply that it is important to be alert to potential trade-offs inherent in the advocacy and adoption of one particular style of thinking and problem-solving. In the context of computation thinking, abstraction is assumed to be 'a friend'. As a critical counter to that assumption, we have considered the respects in which abstraction might be 'an enemy.'

We also take inspiration from this analysis, for future research into novel approaches to end-user programming. Much end-user programming research has implicitly shared the computational thinking agenda, assuming that end-users must develop (or be trained in) the habits of computational thinking in order to construct better end-user solutions. Our previous work on Attention Investment has already identified one respect in which users might have good reasons not to construct abstract descriptions of their task (due to attentional costs, benefits and risks associated with an abstract strategy). However, there are other computational practices that might be advocated for adoption by end-user programmers. The research agenda in end-user software engineering makes this quite explicit, in drawing attention to the various ways of thinking that are of value in software engineering, and looking for ways to instil those professional ways of thinking (e.g. specification, design, testing, debugging) in end-user programmers. We hope to explore an alternative style, in which professional "good practices" are experimentally set aside, allowing end-user programmers to construct programs that are not reliant on abstraction or other central features of computational thinking. Rather than changing the user, assisting him or her to think more computationally, we ask whether computers can be made more accessible to those who, for whatever reason, prefer not to do so.

## 7. Acknowledgments

Luke Church's research is supported by the Eastman Kodak company.

## References

- Anderson, R. (2001). *Security Engineering*. John Wiley & Sons.
- Blackwell, A.F. and Hague, R. (2001). Designing a programming language for home automation. In G. Kadoda (Ed.) *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2001)*, 85-103.

- Blackwell, A.F. (2002a). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.
- Blackwell, A.F. (2006a). Metaphors we program by: Space, action and society in Java. *Proceedings of PPIG 2006*, pp. 7-21.
- Blackwell, A.F. (2006b). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI)* 13(4), 490-530.
- Blandford, A.E. and Green, T.R.G. (2001). From tasks to conceptual structures: misfit analysis. In *Proc. IHM-HCI2001* Vol. 2.
- Bowker, G.C. and Star, S.L. (1999). *Sorting Things Out: Classification and Its Consequences*. MIT Press.
- Brooks, F.P. (1974). *The Mythical Man Month and Other Essays on Software Engineering*, Addison Wesley Longman Publishing
- Brown, B. (2001). Unpacking a Timesheet: Formalisation and Representation. *Computer Supported Cooperative Work* 10, 293-315.
- Carroll, J.M. and Rosson, M.B. (1987). Paradox of the active user. In J.M. Carroll (Ed.), *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, Bradford Books/MIT Press, pp. 80-111.
- Carroll, J.M. (1990). *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press.
- Church, L. (2005). #Dasher: a continuous gesture IDE, In *Proceedings of PPIG 2005*, pp 227-241
- Church, L and Blackwell, A.F. (submitted). Structured Text Modification Using Guided Inference. Full paper submitted to VL/HCC'08.
- Conway, M.E. (1968). How Do Committees Invent? *Datamation* 14(4), 28-31.
- Downer, N., Holmes, N., Sarris, P., Leedham-Green, E., King, F., Stibbs, R., Glendenning, M., Kuczynski, M., Yates, D., Greeves, R. and Pitt, C. (2007). Twelfth Report of the Board of Scrutiny. *Cambridge University Reporter* No. 6082. Available online at <http://www.admin.cam.ac.uk/reporter/2006-07/weekly/6082/17.html> (accessed 15 April 2008).
- Finkelstein, A. and Dowell, J. (1996). A Comedy of Errors: the London Ambulance Service case study" in *Proc. 8th International Workshop on Software Specification & Design IWSSD-8*, (IEEE CS Press), pp. 2-4.
- Finkelstein, A. and Shattock, M. (2001). CAPSA and its implementation: Report to the audit committee and the board of scrutiny, University of Cambridge. *Cambridge University Reporter* No 5861. Available online at <http://www.admin.cam.ac.uk/reporter/2001-02/weekly/5861/> (accessed 15 April 2008)
- House, A.V., Davis, M. Takhteyev, Y. and Ames, M. (2005). The social uses of personal photography: Methods for projecting future imaging applications. In *Proc. Conference on Human Factors in Computing Systems (CHI'05)* extended abstracts, pp. 1853-1856.
- Jenson, S. (2002). *The Simplicity Shift: Innovative design tactics in a corporate world*. Cambridge University Press
- Latour, B. (1987). *Science in Action*. Open University Press.
- McConnell, S. (1993). *Code Complete: A practical handbook of software construction*. Microsoft Press.
- Sewell, P. (2003). *Semantics of Programming Languages*. Course notes for Cambridge Computer Science Tripos, available online at <http://www.cl.cam.ac.uk/teaching/2003/Semantics/>

- Turnbull, D. (2006), The Function of Maps in *Beyond Borders: Thinking Critically About Global Issue*, Paula Rothenberg, Worth Pbls., pp, 7-15.
- Verran, H. and Christie, M. (2007), Using/Designing Digital technologies of Representation in *Aboriginal Australian Knowledge practices*, Human Technology Vol 3 (2), pp. 214-227.
- Wing, J. (2006). Computational Thinking. *Communications of the ACM* 49(3), 33-35.