‘COMPUTATIONAL THINKING’ AND THE POSTCOLONIAL IN THE TEACHING FROM COUNTRY PROGRAMME

Paul Dourish

Introduction

For several years now, my research has attempted to look on computational artifacts not just as things that we build but also as ways of understanding the world, examining the processes by which information technologies are designed and shaped, as they themselves become tools for encountering the world around us. Most recently, this has developed into a project, still in its early stages, that investigates the putative ‘portability’ of design methods in transnational contexts as one part of a broader investigation that we are calling ‘postcolonial computing’ (Irani et al 2010). Information technology design is a global process, and information technology design is often framed, these days, as a site for cultural encounter, frequently in the context of international development (e.g. Best & Wilson 2003; Kam et al. 2007). By calling our research ‘postcolonial computing,’ we want to place computational technologies within the analytic framework of cultural encounter and the historicized global flows of people, power, knowledge, capital, and resources that postcolonial scholars have examined. Design processes in information technology take such categories as users, knowledge, requirements, and representations as givens, but in our research we want to open these up for critical scrutiny, to investigate what work they are doing in transnational and cultural contexts, and to look at how they do them. In the area of Human-Computer Interaction, user-centered and ‘participatory’ approaches to technology design play a central role; the question for our work is what it means to foreground particular views of the ‘user’ or of ‘participation’ when design is a site of cultural encounter. This is the framing device I brought to my involvement with Teaching from Country.

The particular set of questions that I want to explore here pick up on what Nigel Cross (2006) has called ‘designerly ways of knowing’ and what Jeanette Wing (2006) has called ‘Computational Thinking’. Cross and Wing attend to different issues, and are focused on different kinds of

problems, but as I see things there are, in these two formulations, some related ideas that help focus on our engagements with artifacts, and our engagements with the world simultaneously. They help me formulate the contention that the ways in which we create and shape artifacts of different sorts, are epistemological as much as practical. To encounter the world as a place where new kinds of objects and activities can be shaped – not least digital objects – is a way to know the world.

Cross, whose concern is with design in its broadest senses and especially with design education, wants to draw attention to the fact that designers are not simply people who undertake a particular set of activities nor those who produce particular kinds of artifacts, but rather those who have a particular way of knowing the world, one that is conditioned by their designerly stance. The idea of computational thinking is a compelling one, certainly. It moves beyond well-worn arguments about 'computer literacy' to thinking instead about how our interactions with information technology shape our encounters with the world, by focusing on how computer programmers, engineers, and designers might see the world through the lens of information technology. The question is, to what extent are the boundaries and categories of 'computational thinking' set hard? It seems that if users are to negotiate these boundaries and categories they must at least to some extent take on the thinking of designers. The vital questions concern 'the extent of the extent', and actually how negotiations proceed.

Wing is an advocate for the importance of computational understandings in the contemporary world, and his work offers a beginning in attending to those questions. She argues that, in a world in which computational and digital artifacts have an ever more pervasive role, computational thinking plays an important part of contemporary education, alongside mathematical and symbolic thinking, narrative thinking, and those elements that have traditionally played a foundational role in education. Design educator Donald Schön (1983) talks about design as a 'reflexive conversation with materials,' and so clearly the nature of the conversation that one might have is shaped by the nature of the materials with which you might work, and Wing's arguments about computational thinking bring to the fore the question of just what kind of material one is working with when one is working with computational 'stuff.'

In this paper I diffract these ideas through my experience of engaging with Yolŋu people and places and with others involved in the Teaching from Country programme in the course of the short two weeks I spent in northern Australia in July 2009 and through reading the website. Of course my first engagement was making preparations for my presentation to the seminar – supplying a title and an abstract. Uncertain, I simply delayed at first; when I could delay no longer, I sent the most vague title and most general abstract I thought I could manage. I was reluctant to commit to a particular direction, and for good reason. I knew that I would see and hear a great deal during my visit, and that, inevitably, any ideas I put together in the familiar

surroundings of southern California would seem naïve and irrelevant once we were gathered in the seminar room at CDU. What I hoped, instead, was that I would be able to come out, spend a week learning from Michael, John, their colleagues at CDU, the Yolŋu advisors, my fellow international participants, and the others whom we would meet in Darwin and in Gove and, on the basis of all these conversations, put together a new presentation that would synthesize and respond to everything that I had learned.

To a certain extent, this is what happened, but only in part. I had, I think, severely underestimated how much I had to learn about life in the Northern Territory in general, and in the Yolŋu lands in particular. I had, too, underestimated the challenge that this learning would present to my way of thinking about and understanding people, technology, and culture. So, while the paper does indeed respond to things that I learned when I came to northern Australia that July, it touches only on the surface of things I began to understand. Two things that I learned in the first couple of hours after my flight touched down in Darwin came to frame my thinking.

Flexibility is a requirement – that's my first theme. Having been traveling for over 24 hours from Los Angeles to Darwin, I was very much looking forward to a shower, a beer, and a nap (not necessarily in that order) when I got to the guest house where we were staying. But there was a Yolŋu Language and Culture class planned for later that evening; the first class of the semester in the Yolŋu Studies program. I was still hankering for my beer and my nap yet I did not want to miss the class and the opportunity to get started. Here we met Dhanggal and Garnggulkpuy, Yolŋu teachers in the Teaching from Country programme, who for this class were to actually attend the classroom.

The primary focus of this class was a lesson from Garnggulkpuy on freshwater and saltwater and their significance to the Yolŋu people, both in ancestral stories and in everyday life. Freshwater, as she explained, is our source; it is where we come from. Saltwater is where we come together; it signifies negotiation and interaction. Saltwater, Garnggulkpuy explained, is 'where everything takes place.' Through this exploration of freshwater and saltwater, and also through the way that the conversation was conducted, I began to gain a deeper apprehension of the principle of yothu yindi and the relevance of negotiation, fluidity (in multiple ways) and complementarity in Yolŋu thinking. As we began an exploration of these ideas – of the ways that people come together create new spaces of negotiation in interaction, as we mix and combine our individual origins in the creation of collective action, as we understand how individuals and collectives are positioned through our interactions and negotiations – I was, unknowingly, beginning to understand a second theme that would be important for the rest of the trip and, most relevantly here, for some ideas that I want to explore that speak to the relationship between Yolŋu epistemology and ways of understanding the world and the technological contexts in which I conduct my research.

A Story

Let me tell a story to illustrate. It comes from early fieldwork conducted by Lilly Irani, at D-Design, a psuedonymous design firm in Delhi, India. Lilly is one of the colleagues with whom I am involved in a collective struggle to understand what postcolonial computing might become as an analytic that helps us attend to some of the questions I've outlined.

> D-Design were engaged by a development NGO to work on the design of prototype personal water filters. For this study, a team drove hundreds of kilometers searching for locations from which they would seek volunteers from whom they could gather information about needs and practices. The lead designer described the imagined participant as someone 'fairly poor,' getting 'water from the dirty river,' often ill from water-borne illness, and without a filter. What they found instead were villages where people seemed relatively happy or even proud of their water. Complaints of illness were few, though many complained about over-fluoridated water – a problem the clients were not interested in pursuing. Throwing his hands up during on meeting, one of the principals cried, 'Where is the poverty?!' before dramatically throwing his head onto the table.
>
> The team loosened their image of the ideal participant, finding people who were curious enough about the filter and met loosened income requirements. Once villages were selected, the team planned visits to find and screen participants. They planned to interview people in the household, and have them complete collages around themes like 'water' and 'future,' among other activities. However, in much the same way that the notion of their ideal participants had shifted dramatically in the encounter with the field, so too did these methods that the team had hoped to deploy.
>
> The team went to the village to the meet with a man who'd expressed interest in participating in the study. They had planned to interview each person in the household and have each of them complete the collaging activity. However, when they arrived at his house, they found it was actually in the process of being built. The volunteer was living with his mother and his sister under a thatched structure propped up against a tree. For facilities such as water storage and cooking space, he relied on his aunt's house across the small road. Further, the man volunteering was not particularly talkative, which made it challenging to record his thoughts on video. He did, however, have a gregarious cousin at the house across the street. With a little deliberation, the team pulled the cousin into the interview. The individual interviews imagined in the planning had mutated. The aunt's household had been pulled into the project through an ad hoc decision and the talkative cousin. As the time moved on to collaging efforts

> and other design exercises, they soon found themselves a site of much collective village activity and interest, and their pristine ideas about the relevance of their design methods soon had to be radically revised.
>
> Deployed in context, methods and representational practices reveal aspects of their situations of origin, and frequently carry their cultural assumptions with them in ways that can be problematic. The language of the design brief to which D-Design was responding was the traditional language of the discipline of Human-Computer Interaction – stakeholders, usability, and requirements. This language reflects a conceptual framework or infrastructure within which the encounter between design practice and everyday life is framed. Designerly ways of knowing are framed here as the ways that one can know about the world through the deployment of particular kinds of design practices – practices that may or may not, as we see in this example, successfully escape the contexts of their own production.

The particular set of design practices that motivate me, and which are fundamental to Teaching from Country, are the design practices associated with information technology and digital media. Arguably, nothing is more fundamental to the production of information systems than the computer programs that comprise them, and by extension, the programming languages in which those computer programs are written. These programming languages are the formal expressions of a computer system's behavior through which programmers and engineers create new software systems. Often, when we think about information systems and their impacts, the actual practice of programming disappears; we think about the contexts (organizational, institutional, economic, political, and historical) within which software systems are developed, we think about their ramifications and implications for infrastructure, training, and literacy, and the co-evolution of software systems and daily practice, but the actual lines of software core, written by some set of people sitting at a keyboard somewhere, withdraw into the background.

However, every bit as much as the software systems that they describe, the programming languages and programming systems themselves comprise an important resource for understanding, encountering, exploring, and representing the world, and they deserve some serious examination (something which the 'software studies' movement of recent years has finally begun to do; see, e.g., Fuller 2008). I want to turn now to some discussion of the computer programs and programming languages that make up software systems as a way to examine some opportunities that Teaching from Country opens up in this domain. To do so, I need to begin by setting out a brief introduction to the material of computer programs, for those who are unfamiliar with the practices of programming.

The Material of Computer Programs

Computer programs generally comprise large bodies of text. The text is what is often known as 'source code' – expressions and statements constructed according to the rules of particular programming 'languages.' A small program might consist of a few tens or hundreds of such lines of text; a large program might comprise millions. Some lines of text define data objects that the computer program uses to represent the world such as the records that might describe people in a social networking system, video streams in a videoconferencing application, or web pages in a tool to help you create a website. Other lines of text define the operations that might be performed on those data objects such as looking up a person's friends, initiating a video stream from one computer to another, or printing out a web page.

One of the fundamental questions that programming systems need to solve is, how should these lines of text – these computer instructions – be organized? Over the decades that people have been building computer programs, a few different styles have emerged. For instance, we could completely separate the two – we could keep the 'data' and the 'operations' separate. Early programming languages often worked this way. A popular arrangement in more recent years has been to combine them in particular ways. In one style, called 'object-oriented programming' (or OOP), we combine the data element with the specific code that operates on it (rather than on other elements), and the combination is called an 'object.' In object-oriented programming, these two are so tightly combined that we don't think of processing a data element using some procedure; instead, we think of 'asking the object to perform an operation on itself' (because the object has the procedural code built into it.) The blocks of source code that define how an object should perform some operation are called 'methods'; the requests that objects might send and receive, which ask them to perform particular operations, are called 'messages.' So, in a particular software system, I might send a message 'print yourself' to an object that represents a web page; it would find the method that it knows for responding to this message, and perform the operations. One advantage of this arrangement is that when I send a message 'print yourself' to a different kind of object – one that represents a person, or a data file, or a video stream, or a person, then they might all behave differently, in much the same way that, as human beings, if we are asked to do something, we might do it in different ways depending on who we are or what kind of role we have.

These 'kinds' of objects are called 'classes.' When I build a system using object oriented programming, then my lines of text describe the classes of objects that my system will need to use ('Friend', 'VideoConnection', 'WebPage') and then the methods that objects of each class will need so that they can operate effectively when they receive messages.[1] Once I have done

1 There are many different object-oriented programming languages, which embody different ideas and employ different terminologies. Here, I am using the terminology employed in the early and highly influential Smalltalk language (Goldberg and Robson, 1983), although the same ideas occur in one form or another in most object-oriented languages.

that, as a programmer, the system goes into operation, and the fundamental rule that applies is: how an object responds to a request for action (a message) depends on what sort (class) of object it is.

The outline above is necessarily very sketchy, but it should provide non-programming readers with an orientation to the basic ideas and conceptual structures that software developers are manipulating in the creation of (at least some) software systems. It also provides a starting-point for exploring a couple of alternatives. While the idea of object-oriented programming is in wide circulation, these two alternatives are much less widespread; they constitute interesting ideas that have been proposed in research papers but have not by any means made their way into mainstream software development. However, they are useful tools for reflecting upon the ideas embodied in object-oriented programming and some potential revisions to 'computational thinking.'

Two Alternative Accounts

The *first* of the alternative ideas is called 'subject-oriented programming' (Harrison and Ossher 1993). The central idea behind subject-oriented programming is a very simple extension of the basic principle of method discrimination in object-oriented programming, as I described it above. The idea is this: when a message is sent to an object, the method that will be executed depends on the class of the object receiving the message (as in traditional OOP) *and also on the class of the object that sent the message*. The basic idea here is that the kind of response that an object might make to a message depends on the kind of object that sent the message in the first place. Again, this is a very familiar idea in everyday life; how you answer a question, for instance, depends not just on your circumstances but also on the person who asks you (a friend or family member, a stranger, a policeman, a child, and so on). It is not appropriate to draw too much on these analogies between program behaviour and human or social behavior, of course, but to the extent that programming systems are used to build models of the world, and to the extent that what they capture is something that Wing and others label as 'computational thinking,' then it is important to recognize that subject-oriented programming is fundamentally a relational way of modeling action and an interactional way of accounting for emergent behavior in a system. It captures a dimension of expression that is not present in object-oriented programming or in traditional algorithmic thinking, and as such, when we think of computer programs as representational schemas through which programmers and engineers encounter the world, it provides conceptual resources for understanding relational phenomena.

The *second* alternative idea that I want to discuss is called 'predicate classes' (Chambers 1993). In traditional OOP, when an object is created, it is always created with a particular class. (In most OOP languages, the way to create a new object is essentially an expression that says

'make a new object of class Thing'; in other words, there is no way to create an object without specifying its class.) The class of an object, in almost every OOP language, is fixed; once a Thing, always a Thing. Similarly, the relationship between classes is fixed; one class might be a subclass of another (a more specialized kind of object, the way that Table might be a more specialized subclass of the general class Furniture), and that relationship will hold for as long as the system operates. Predicate classes, though, are slightly different. A predicate class is defined by two things; a class to which it is related, and a rule that specifies when an object is a member. For instance, I might create a class 'Square' by specifying that a Square is a kind of (a subclass of) Rectangle, and that a Rectangle is a Square whenever two adjacent sides have the same length. (These are called 'predicate' classes because a predicate is a computational expression whose value is either 'true' or 'false.' The number '4' is not a predicate, nor is the string 'Paul,' but the expression 'do adjacent sides have the same length?' is either true or false for any particular Rectangle.)

Predicate classes create an interesting new opportunity for OOP systems. Now, the class of an object is not fixed; it depends on circumstances. New methods and new behaviours might be associated with an object when it becomes a member of the predicate class; but when the circumstances change once more, they no longer apply. Similarly, in conceptual terms – that is, within the frame of computational thinking – it provides us with a means of seeing the world in dynamic, contingent, and circumstantial terms.

In subject-oriented programming then, the basic execution model of object-oriented programming has been shifted so that it takes a relational stance. The behaviour of an object is not simply a question of its identity, whatever that should be; rather it is a product of the interaction of objects, or a product of the particular configuration of message sender and message receiver. While the computational changes are small, the interactional consequences are significant, and perhaps more to the point, the representational practices at work in creating a system that operates this way open themselves up to an alternative epistemological foundation. A similar small-but-significant shift is at work in the example of predicate classes, then, which replace a notion of fixed identity with one of contingent identity; the roles that objects play, the behaviours available to them, and so on, are subject to continual reassessment and reconsideration depending on circumstances.

Using these Insights to Think about 'Computational Thinking'

Let me take a step back and think about these ideas in terms of arguments about 'computational thinking.' The argument around computational thinking is, essentially, that there is some particular way of approaching the world that is particular to computer science, and, conversely, that there are some particularly useful ways of thinking about the world that computer science might

offer. Computational thinking suggests that particular modes of theorizing, such as algorithmic specification and procedural abstraction, offer important intellectual tools for understanding the world, with a particular emphasis upon the opportunities for digital representation, but not solely with that end in mind. What I think these two examples, subject-oriented programming and predicate classes, start to illustrate is that there is perhaps no one unique model of 'computational thinking' but rather that computational tools embody, and provide a platform for thinking about, different epistemological approaches. The fundamentally relational perspective at work in subject-oriented programming, and the fundamentally contingent perspective at work in predicate classes were things that I was strongly reminded of as I heard people talk, tell stories, describe spaces, interact with each other and with others in Arnhem Land; and what it made me think was not, 'Oh, computational thinking has already encompassed these,' but rather, 'what a marvelous vehicle these kinds of computational tools might be for discovering (rediscovering?) and exploring this sort of world-view.' Computer programs and programming languages are not just tools for getting work done; they also shape how we think about the world (where 'we' means computer programmers, engineers, scientists, and in turn, to an extent, the users of computer systems). This is computational thinking, too. The two ideas I have been outlining – reminiscent as they are of some of the themes that arose in my first few hours in Darwin and then later in Arnhem Land – show that aspects of Yolŋu epistemology highlight what computational thinking might be, what it might do for us, and what opportunities it might embody.

It would certainly be absurd to claim that building software systems with these tools will result in systems that are inherently culturally appropriate and responsive. Anyone can build bad systems with good tools. However it might be more useful to think here in terms of computational thinking. If the tools that we provide for modeling, encountering, and framing the world are ones that are based on the importance of relations between people, of the contingencies of those relations, of the importance of responsiveness to local circumstance and immediate need, of the relevance of relations between interactional participants in the moment, in the future, and in the past, then we are perhaps opening up a fascinating area of potential impact from a project like Teaching from Country. Participants in Teaching from Country have been concerned with crafting tools and technological environments that can be effective for Yolŋu teaching, but it the project opens up, I believe, broader questions about the potential relationships between technology and cultural practice in ways that can enrich and enliven our ideas of 'computational thinking'.

The View from Arnhem Land

While we were on country, Dhanggal took us to Galuru; we ate lunch there on the first day of our visit to the Gove peninsula. After lunch, I walked around for a while on the beach for a while, just getting a feel for the place. Galuru, where the billabong runs to the ocean is, of

course, a place where the freshwater and the saltwater meet. It is also a place that, I realized as I walked, must change drastically as the rains come in the wet season, as the creek floods, as the tides shift. The long sand flats were wet and soft in some places, dry and firm in others, with pools of water from where the shifting currents had left it. At the same time as it embodied all this change, though, Dhanggal was telling us stories of her visits to the place as a child, and its importance to her ancestors, and across all of this, of course, it remained the same place. As I walked, it seemed to me to embody a fascinating encounter between stability and change. By the same token, it's very easy to talk of technology as something that changes rapidly; and yet some of the ideas that it embodies are very old ones. It is useful to examine what kinds of opportunities for change might yet be embedded within the worldview from which information technology springs. Arnhem Land is a fascinating place to start to do just that.

Acknowledgements

I owe enormous debts of gratitude to Michael Christie, John Greatorex, and their colleagues at Charles Darwin University for the opportunity to participate in the program; to Garnggulpuy, Dhanggal Gurruwiwi, Gotha, Yingiya Guyula, and the other Yolŋu participants in the Teaching from Country programme, for patient education; to other participants in the seminar, for inspiration and critique; and to Helen Verran for making initial contacts and for thoughtful editorial intervention. This work is supported in part by the US National Science Foundation under awards 0712890, 0838601, 0838499 and 0917401.

References

Best, ML & Wilson III, J 2003, 'A growing community of interest', *Information Technologies & International Development*, vol. 1, no. 1.

Chambers, C 1993, 'Predicate classes', *Object-oriented programming, European conference proceedings ECOOP'93,* Kaiserlauten, Germany, pp. 268-296.

Cross, N 2006, *Designerly Ways of Knowing,* Springer.

Fuller, M 2008, *Software Studies: A Lexicon,* Cambridge, MIT Press, Massachusetts.

Goldberg, A & Robson, D 1983, *Smalltalk-80: The Language and its Implementation,* Reading, Addison-Wesley, Massachusetts.

Harrison, W & Ossher, H 1993, 'Subject-oriented programming: a critique of pure objects', *Object-oriented programming systems, languages and applications ACM conference proceedings OOPSLA'93*, pp. 411-428.

Irani, L, Vertesi, J, Dourish, P, Philip, K, & Grinter, R 2010, 'Postcolonial computing: a

lens on design and development, *Human factors in computing systems, ACM conference proceedings CHI 2010*, Atlanta, Georgia.

Kam, M, Ramachandran, D, Devanathan, V, Tewari, A, & Canny, J 2007, Localized iterative design for language learning in underdeveloped regions: the PACE framework*', Human Factors in Computing Systems, ACM conference proceedings CHI'07*, San Jose, California, pp. 1097-1106.

Schön, D 1983, *The Reflective Practitioner: How Professionals Think In Action*, Maurice Temple Smith.

Wing, J 2006. 'Computational thinking', *Communications of the ACM,* vol. 49, no. 3, pp. 33-35.