Wednesday 5th of September

12:00 - 14:00  Registration open
14:00 - 14:05  PPIG Open  
  Welcome to the ‘Art Workers’ Guild’  
  Prue Cooper
14:05 - 14:50  Keynote  
  Charlie Gere
14:50 - 15:15  Coffee Break
15:15 - 16:30  Recital of Postcards  
  Tangible Workshop
16:30 - 17:00  Coffee Break
17:00 - 18:15  Session 1  
  Recital of Postcards  
  Invited talk  
  Beach-combing lumps of geometry from the sea shore of Mathematics  
  Fred Baier  
  A Craft Practice of Programming Language Research  
  Alan Blackwell
19:00  PPIG Dinner at The Lady Ottoline  
  Address: 11A Northington St, London WC1N 2JF

Thursday 6th of September

10:00 - 11:00  Session 2  
  Invited talk  
  The Multiple Psychologies of Programming in the Service of Learning Science and Mathematics  
  Andrea diSessa  
  Recitation of poems  
  Nicholas Cooper
11:00 - 11:30  Coffee Break
11:30 - 13:00  Session 3

The Lish: a data model to support analysis by end user programmers  
Alan Hall  
14

Modeling cognitive processes underlying computer programming  
Catalin Florian Perticas and Bipin Indurkhya  
16

Direct Programming  
Jonathan Edwards

How do people learn to use spreadsheets? (Work in progress)  
Advait Sarkar and Andrew D. Gordon  
28

13:00 - 14:00  Lunch

14:00 - 15:30  Session 4

Recital of Postcards

How I Use Computers  
Simon Hurst

Choosers: The design and evaluation of a visual algorithmic music composition language for non-programmers  
Matt Bellingham, Simon Holland and Paul Mulholland  
36

Invited talk and demo  
Valaa Open System (ValOS)  
Ville Ilkkala

Hands on with Boxer  
Andrea diSessa

15:30 - 16:00  Coffee Break

16:00 - 17:00  Session 5

Recital of Postcards

All recordable human discourse is trapped in aleph-zero  
Oli Sharpe  
46

What Lies in the Path of the Revolution  
Antranig Basman  
55

17:00 - 18:00  Open Panel Discussion

19:00  Musical Soirée (bring your instrument!)
Friday 7th of September

10:00 - 11:00 Session 6

The Logos in Computation and Everyday Life
James Ramsay

A growing tip or a sprawling vine: how software grows
Luke Church and Mariana Marasoiu

Wide, long, or nested data? Reconciling the machine and human viewpoints
Alan Hall, Michel Wermelinger, Tony Hirst and Santi Phithakkitnukoon

11:00 - 11:30 Coffee Break

11:30 - 13:00 Session 7

Recital of Postcards

Investigating multimodal affect sensing in an Affective Tutoring System using unobtrusive sensors
Hua Leong Fwa and Lindsay Marshall

Conjuring Code
Will Houstoun and Marc Kerstein

Explicit Direct Instruction
Felienne Hermans and Marileen Smit

Phenotropic Programming?
Clayton Lewis

13:00 - 14:00 Lunch

14:00 - 15:30 Session 8

Recital of Postcards

Investigating Conversational Programming for End-Users in Smart Environments through Wizard of Oz Interactions
Kate Howland and James Jackson

Crafting Design Documents in First-Year CS Courses
Shannon Ernst and Jennifer Parham-Mocello

On Continuing Creativity
Colin Clark and Sepideh Shahi

15:30 - 16:00 Coffee Break

16:00 - 16:45 Keynote
What if software were different?
Clemens Klokmose

16:45 - 17:15 PPIG Close
Abstract
It has become increasingly common to consider programming as a craft practice, driven largely by tools that are sufficiently responsive for reflective conversation with material, agile design processes, and live coded performance. This paper considers some of the epistemological questions that arise in programming as a critical technical practice, and especially when programming language research itself is taken seriously to be a craft.

1. Introduction
As part of a 2010 investigation of software development practices among digital arts professionals, Woolford et al (2010) interviewed Rosy Greenlees, the director of the UK Crafts Council, in order to identify ways in which critical understanding of quality in artistic software development might be informed by craft practice traditions. Now is a good time to reflect on that investigation, in a year when the annual Psychology of Programming Interest Group meeting is hosted by the Art Workers’ Guild.

We can continue to draw on Philip Agre’s concept of a critical technical practice, originally introduced in his notes on attempting to reform AI (Agre 1997). Agre was concerned with the implications of research that is necessarily both philosophical and practical – involving technical construction alongside a discourse that analyses and describes the thing being built. True AI research always involves both elements (it is possible to talk about future AI systems without building them, but that approach is more commonly associated with science fiction than with science).

The study by Woolford et al was primarily concerned with criticality in artistic practice – how is the intellectual discourse of the arts shaped in relation to philosophical, social, political or other concerns? However the software developers interviewed in the course of that project, despite working in a professional arts context, emphasised the practical engineering disciplines inherent in their work as being primary. Unlike Agre’s AI researchers, the principal outcome of their work is an artefact. Engineering is essential in their business because, above all else, the show must go on.

When meeting at the Art Workers’ Guild, how might programming language researchers situate the technical and philosophical aspects of their investigations? Is a programming language primarily an idea, to be instantiated in mundane technical implementation, or is it an artefact whose construction leads us to understand what it is – in other words, a craft? Programming languages are closely associated both with AI and with digital arts, so is programming language research an art or a science?

Edsger Dijkstra (1977) famously rejected the notion that programming was a craft – or rather, he acknowledged that it was practiced as a craft, but argued that it should not be, that it must become a science. Nevertheless, contemporaries such as John C. Reynolds, in his formally-motivated textbook The Craft of Programming (1981) continued to appeal to the notion of craft as embodied skill for the expert practitioner. Antranig Basman (2016) playfully reflected on these distinct notions in his essay exploring the virtues of craft practice, turning the title of his PPIG paper into a footnoted lament that Building Software is Not “[yet] a Craft.”

In my own experiments in programming language design, I have explicitly considered whether a programming language can itself be produced through a craft practice (Blackwell 2013, Blackwell & Aaron 2015), and have created an experimental language through this process (Blackwell 2014). There are, of course, many different practices that can be brought to software projects (Bergstom & Blackwell 2016), and my own experimental production of the Palimpsest language is only one of
these. Nevertheless, as a distinctive methodological approach, it is particularly appropriate to reflect on its implications at this Art Workers’ Guild meeting.

Notions of craft, extending beyond skilled expertise (the titular sense of Reynolds’ proof-based programming textbook), to the embodied knowledge of a tool held in the hand, have become increasingly current in discussion of programming as tools become increasingly live (Tanimoto 2013). Facilities that accelerate programming through code completion, text prediction, refactoring support and so on mean that the programmer’s intentions flow through the typing hands even more quickly than read-eval-print loops or interpreted environments such as Smalltalk. The agility of working in (and fluently modifying) the Smalltalk environment has already had profound effects on the software industry, and indeed on digital experience universally, not only through the interaction modes of the GUI (Blackwell 2006a), but in the live collaborative editing practices of the wiki (Leuf & Cunningham 2001), the design understanding of the pattern language (Blackwell & Fincher 2010) and the managerial philosophies of SCRUM and other agile development practices (Beck et al 2001), all of which emerged from the Smalltalk community.

But the question asked in this paper is whether these live and embodied craft practices can be turned back, not simply in the professional environment of agile software engineering, or the creative context of live coding performance, but as an element of programming language research itself. Can the commonplace analogy of the apprentice making her own tools be appropriately applied to the craft-making of a programming language as a tool?

2. Experiencing Software Craft

Code, as observed by Daniel Cardoso-Llach (2015), carries metaphorical associations of weightlessness. Codes are disembodied languages rather than physical machinery. Yet codes are also rule-systems and orderings, specifying regularities over the material world of substance and phenomena. For example, performance coding as an artistic practice imposes order through sound and light fields, just as the sculptor's chisel extracts form from stone, or the painter's brush arranges it from pigment. Of course, although software is conceptually abstract, it has always also been wholly embodied. The computers in which these codes are constructed, executed and observed are complex and costly assemblages including rare minerals and sweatshop labour. Their operation depends on a material infrastructure of communications, power networks, server farms and processor clusters spanning the globe.

The tension between the materiality and immateriality of code constantly challenges the ways in which we understand it. As an abstract mechanism of control, code has traditionally been an instrument of government, relieving the military-industrial complex from the uncertainty of human workers, replacing fallible or undisciplined human hands with the precisely replicable actions of machines. Computer-aided design and manufacturing, 3D printing, and software itself, are engineering intermediaries between the industrial body and the governmental soul.

However, when live-coders turn code into a medium for artistic exploration, we overturn much of this conventional understanding. Rather than an instrument of control, through which engineers impose order on a chaotic world, code itself becomes a material within which the craft-coder develops and displays a creative practice. Code is not the chisel, but the wood. There is much to learn from this new ontology of code - both in relation to the nature of technology, and in relation to the nature of practice.

This change is occurring as those parts of the software industry concerned with user experience and interaction design have had their attention drawn away from the traditional office workstation with its visual display screen and typewriter keyboard, to the many forms and contexts in which digital processors are now found (Wiberg 2013). Where software was once separate from the materiality of everyday life, sustaining a kind of technological mind-body dualism, we have now become thoroughly entangled with computers that are embedded in our clothing, our cars, our chests, our pets, or attached to our wrists and on our faces. After losing their screens, embedded computers become Tangible User Interfaces, joining the Internet of Things.

Interaction designers for this tangible, embodied and embedded world of computation are thus re-engaging with materials and craft practices in order to build their interactive metal, wooden or cloth
prototypes. Rather than working with the "pure digital" (Hanse et al 2014), they have again become craft makers. And as reflective practitioners, this creative design research work draws their attention to the resultant conversation with materials that is familiar in the design theory of Donald Schön (1983). The user experience research literature delights in this turn to a new materiality, because of the way that it offers insights from more established branches of design research (Gross et al 2013).

Unsurprisingly, this research engagement with new-found material practices has also led to a concern with the materiality of code itself. Not all writers take the analogy this far, but many interaction designers perceive their experiences with the software of their prototypes as having a great deal in common with their rediscovered experiences of hardware. They feel that, even after turning from the workbench back to their laptop keyboard, they are still having a conversation with a material (Schön 1983), in which it resists their intentions, disrupting their pure theoretical conceptualisations via the mangle of practice (Pickering 1995).

Coding is indeed hard, and code often seems to be resistant to the intentions and desires of the coder - experienced in much the same way as when physical materials resist craft labour. But if code is a material, it would appear to be a surprisingly immaterial one. The simultaneous immateriality of code means that it is equally resistant to this alternative characterisation by design theorists. Surely code cannot be material in the same sense as a plank of wood, or a ball of clay, which require (as Sennett describes) a dialogue between the head and the hand? Yet interaction design theorists persist in the argument that software is material, and that where there is a material, there must be a craft. Programming is described as a craft skill, with practitioners writing manifestos for "software carpentry" or "software craftsmanship". Even Sennett describes open source software development as "public craft".

Once again, this presents a challenge in drawing the appropriate analogies between our traditional understanding of craft and materials, and the experiences of making software. McCullogh's celebration of the "practiced digital hand" (1998) describes a master user of computer-aided design tools as engaged with coaxing reluctant or recalcitrant digital materials. Gross et al (2013) draw on Cohen's theory of artistic media to explain why this very recalcitrance becomes a media resource for performance and exhibition, wherein audiences appreciate the virtuosity that has been exhibited in the struggle with a "viscous" medium.

The reader may be thinking that the matter-mind dualism implicit in these distinctions and discussion is either unwarranted or unsophisticated. Perhaps this problem results in part from the need for a new and more subtle conceptualisation of computation as extended cognition. Just as the 'pure' code of theoretical computer science is actually embedded in large material infrastructure, so craft practices are physically embodied in the craftsperson, and socially embedded in communities of practice.

One such long-standing community of practice is the demo-scene, an antecedent of the live-coding community that shares many common concerns with live coders. Demo-scene participants create virtuoso technical artworks, which they present to their peers in competition and performance events. Hansen et al (2014) undertook an ethnomethodological study of the demo-scene community, from which they developed a theory of craft practice, as observed among these code-artists. They see a relationship between the rhythmic elements of the artworks, and the rhythmic practice of tweaking and refining code. In their analysis this material practice results in a craft skill, moulding the practitioner at the same time as the material, developing technique as the basis for creative expression.

But should we expect the audience experience of a product to be the same thing as the experience of making it? We may admire the determination of the demo-scene perfectionist, tweaking his assembly code until every aspect of the sound and imagery are synchronised, but this repetition is surely not the same thing as the execution rhythms of the product itself.

Tim Ingold (2010) offers us an alternative conception of craft knowledge, in which there is no repetition (only machines repeat mindlessly), but rather one step after another, along a journeying path. The craftsman's tool seeks and responds to the grain of a material, in a process of accommodation and understanding rather than imposing form on inert substance. Material should be considered as 'matter-flow', in flux rather than stable, and the craftsman follows the material, in a
manner that Deleuze and Guattari have described as itineration, rather than iteration. The craftsman is thus an itinerant wayfarer, whose practice is one of journeying with the material.

Ingold's work provides one of the most productive perspectives in contemporary discussion of materiality, and offers an ideal analytic perspective for the live coding situation. He applies Alfred Gell in identifying a kind of mistaken belief, in which an object is taken to be the starting point for an enquiry that traces backwards from the object to find the conditions and creative agent that caused it to exist. The object becomes a static index of a prior causal chain, rather than a thing unfolding through the interaction of a maker via the flows and forces of material. The alternative process-oriented perspective of flow and unfolding is unfamiliar to many technologists, but familiar to the contemporary artist, for example as expressed in Paul Klee's classic evocation of drawing as 'taking a line for a walk.' It resonates equally well with the experience of the live coder, who is engaged in a process of programming, but with no intention to create a software product.

Ingold himself observes how different these material craft practices are from the world of technology. He describes technology itself as being an ontological claim. The claim of technology is that things come into being through the application of rules and rational processes, and that objects are thus formed out of inert and undifferentiated substance. If this is true, then surely code, as a rational rule system beyond all others, must be preeminently technological, and certainly not a craft material?

There are still computer scientists who resist the suggestion that computing might be a craft (Lindell 2014). The tension is so long-standing that even Babbage engaged in long-running dispute with Clement, the engineer building his Difference Engine, who claimed that he, rather than Babbage, should be recognised as its inventor (Cardoso-Llach 2015). Computer science is the domain of the gentleman academic rather than the rudely mechanical engineer, and its highest aspiration is to prove the correctness of its products in the manner of a mathematical theorem. Dijkstra's regret of the tendency for software development to be treated as a craft, rather than an automated and repeatable scientific discipline, follows in this line.

Nevertheless, the everyday professional practices of 'agile' software development, like the creative practices of the live-coder, seem far more fluid than a desire for rigorous formality might suggest. Agile developers respond to events, rather than simply following a plan. Their practice, as with Suchman's situated cognition (1987), demonstrates the contingency of rational action, in which the rational agent improvises and adapts to the world rather than imposing order on it. In practice, code seldom attains the mathematical standards that theoretical computer scientists aspire to. The practice of live coding, in which code is a process to be experienced rather than an intermediate specification accounting for an indexical product, is indeed a craft.

So while theorists of materiality in interaction design might argue that software is a design material like their other materials, and that where there is a material there must be a craft (Lindell 2014), an understanding of live-coding takes us in the reverse direction. Following the analyses of Ingold and Sennett, software construction is a craft - and given this craft, it seems that code must be its material. Its materiality arises from its fluidity. Through code, it seems that we have made language into a material, even though this material is insubstantial. Perhaps this is completely appropriate in an information economy and media society, whose products and commodities have also become insubstantial.

Furthermore, the 'conversation with materials,' that has been observed in craft and design practice by theorists such as Sennett and Schön, now becomes a more literal conversation composed of 'linguistic' (or at least notational) exchanges. The regularities and explicit observability of code notations mean that we can more readily understand the patterns of experience inherent in such craft, reflecting on those experiences in the form of pattern language (Blackwell 2015). We can also appreciate a diversity of craft practices, extending beyond live coding to other communities of practice and other practices of programming (Bergström & Blackwell 2016).

3. Manipulate/Automate/Compose

My own research in creating the Palimpsest language deliberately followed an unconventional design process, as a strategy intended to generate novel approaches to existing problems. In particular, it was
motivated by the creative design practice of “letting the material take the lead” (Michalik 2011), as elaborated in the previous section. Although the analogy between software and other “material” was always going to be problematic, there were two respects in which that strategy appeared feasible at the time.

The first was that reliance on the craft tradition embodied in specific professional tools allows the craft designer to carry out technically competent work without conscious critical intervention that might otherwise be an obstacle to innovation. In the context of Java development, this tool-embodied knowledge was obtained through the IntelliJ IDEA environment. IntelliJ has many productivity-assistance features that are clearly based on expert programmer practice, but integrated directly into the editor rather than via dependency analysis and structuring tools. Syntax-directed completion of identifier names, based on combined use of compiler symbol table, library data, and use of an English dictionary to recognize semantic structure in camel case identifiers, meant that most code was both rapid to enter, and correct at the time of entry. Large-scale refactoring tools, mainly providing syntactically-aware renaming of identifiers spanning filenames, class declarations and instance identifiers, made it possible to work fluidly with an evolving conceptual model of the system architecture and functionality. These development features have been introduced to professional tools through the demand for agile development practices, but this project suggests that they have also become an academic resource within a design research context.

The second point of comparison came from classic literature exploring “conversation with the material” from a cognitive perspective (e.g. Schön 1983, Schön & Wiggins 1992), through which design concepts would be refined by observing the development of a sketch or model. A standard account from design ideation research is that sketches are intentionally produced in a manner that allows ambiguous readings (Goldschmidt 1999), so that the designer may perceive new arrangements of elements that would not be recognized in verbal descriptions or internal mental imagery (Chambers & Reisberg 1985). In the case of the Palimpsest development, this “conversation” was achieved through using the system in development, not by systematic functional testing (although detection of bugs was an added benefit), but exploration of the artistic potential of the system. Since Palimpsest itself offered an alternative conception of programming, these experiences often provided insight into the system architecture and development process, not simply the user functionality.

I was concerned that to some extent, these craft practices simply resembled undisciplined programming – hacking, in the old terminology – as opposed to professional software design development. It is true that this project, as with much academic software development, was carried out using methods that differ from professional practice. However, as with academic research, there are also essential aspects of discipline that are a prerequisite of success. The first is an awareness of relevant theory and design precedents that inform an original research question (e.g. systems such as Sketchpad, Smalltalk, Garnet, Visicalc, Toontalk, Scratch and others), while the second is rigorous reflection on the work in progress. This need not be conducted in the manner of an engineering investigation – indeed, the most valuable findings were insights that occurred after periods of relatively unconscious reflection, and in informal journal entries (Blackwell & Aaron 2013).

However, the design intentions for the Palimpsest language were also strongly influenced by the earlier investigations that I had carried out into the cognitive demands of end-user programming (Blackwell 2006b, 2013). In particular, my theoretical approach to support of end-user programming was motivated by the Attention Investment model of abstraction use (Blackwell 2002). Although this is a cognitive model oriented toward design analysis, it does not directly offer design recommendations. Previous work by Wilson, Burnett et al. (2003) had operationalized the attention investment model in a specific design strategy for end-user debugging that they describe as “Surprise, Explain Reward”. One objective of the Palimpsest project was to identify further concrete design guidelines of this kind.

Much of the Palimpsest design was motivated by the need for smooth transitions between direct manipulation and the definition of abstract behavior. This supported exploratory artistic practices, and also avoided the negative consequences of the Cognitive Dimension of *abstraction hunger*, as exhibited by many programming languages and tools. Reflection on development and use of
Palimpsest made it clear that there were in fact two transitions in the level of abstraction provided during system exploration. The first was between direct manipulation of a control on a Palimpsest image layer, and indirect manipulation of the same parameter via a value layer. The second transition is the composition of the behaviors created using value layers and references, by collapsing into collections, by copying, or by modifying the references with indirection layers.

This chain of attention investment transitions from direct manipulation to more complex automated functions and then to scripts had previously been explored in the context of the tangible programming system Media Cubes (Blackwell & Hague 2001), where domestic remote control buttons were used as tangible representations of the actions they controlled. The design intention, as in the Piagetian approaches to teaching mathematics or programming, was that users could build confidence in the physical elements of the representation through familiar concrete operations, using them as simple remote controls. Once those unit operations had become sufficiently familiar (perhaps over a period of months or years), the physical objects would naturally start to be treated as symbolic surrogates for those direct actions, and then used as a reference when automating the action (for example, setting a timer to invoke the relevant action). Once the use of references had become equally familiar, the user might even choose to compose sequences of reference invocations, or other more sophisticated abstract combinations.

In homage to the Surprise, Explain, Reward design strategy developed by Burnett’s group, this approach to the transition between direct manipulation and programming functions can be described as Manipulate, Automate, Compose. The user is able to achieve useful results, and also become familiar with the operation of the system, through direct Manipulation that provides results of value. The notational devices by which the direct manipulation is expressed can then be used as a mechanism to Automate them, where the machine carries out actions on the user's behalf. Finally, all of the functions that the user interacts with in these ways can be Composed into more abstract combinations, potentially integrated with other powerful and/or complex computational functions.

In the case of Palimpsest, it was visual representation rather than tangible representation that was used as the familiar concrete element. Palimpsest was aimed at users who, although they may prefer not to use text in their work (cf Church et al 2012), are comfortable and expert in the use of pictorial representations. By applying minimal constraint on the content of those pictures, the system supports engagement of the same kind provided in drawing and painting tools. The facilities to automate and compose direct manipulations of the image are not imposed on the user at the outset (the Cognitive Dimension of abstraction hunger), but are available for use at any time (abstraction tolerance).

This design approach could, in principle, be applied to any interactive system that offered scripting or extension capabilities. In practice, most software products do not offer users the second two steps in this skill-development process - GUI users are expected to remain at the "Manipulate" phase, and are given little encouragement to move on to Automating and Composing - precisely the points at which computers offer real labour-saving potential. Programming by Demonstration systems (Cypher 1993) aim to facilitate transition from Manipulate to Automate, while Programming by Example systems (Lieberman 2001) uses additional inference methods to Compose automated functions over a range of invocation contexts. All of these transitions can in principle be systematized and characterized in terms of Tanimoto’s 5th and 6th levels of liveness (2013).

It is interesting to note that theoretical approaches to programming language design generally proceed in the opposite order - the mathematical principles of language design as presented in programming language design textbooks (e.g. Finkel 1996) are fundamentally concerned with composition. The syntactic features of the language are recognized as having implications for usability, even though they may be “syntactic sugar” from a mathematical perspective. Once the mathematical and syntactic properties of the language are established, interface libraries provide facilities to support external functionality such as disk, network or graphics output. Finally, a programming environment is defined, in which the user is able to manipulate the syntactic notation to controls these elements. After a language has been in use for a while, live debugging environments might even provide the ability to directly manipulate objects of interest from the user domain, within the context of program...
development. The Manipulate/Automate/Compose strategy of Palimpsest is thus one starting point for a craft-oriented and user-centred, rather than computationally-centred, programming language design.

4. Build/Reflect/Notate - a Process for Crafting Languages

Despite the formal and abstract presentation of many programming language design textbooks, most programming language designers are motivated by a need to change user experience of programming – usually based from introspection on their own experience, or perhaps (if academics) from observation of conceptual problems faced by their students. As a result, the great majority of programming languages are designed for people who are already programmers. Research in end-user programming attempted to address this deficiency through the application and adaptation of user-centered HCI research methods for the programming domain – including development of user models, techniques for analytic evaluation and critique, and a mix of contextual and controlled user study methods.

The Palimpsest development explored an alternative approach, derived from the methods of practice-led design research. It relied on the availability of a new generation of “craft” tools for agile software development, enabling conceptual advance to be made in the context of prototype construction. Almost every aspect of the Palimpsest design, including its conceptual foundations, developed through reflection on the experience of building the system. The research outcomes can be attributed in part to insight and serendipity, in a manner that while probably recognizable to many research scientists, is not normally specified either in prescriptions of scientific method, or of user-centred software development methodology.

Apart from the craft tradition and tools that enabled discovery through construction, the central focus of the project was to understand a class of potential user (the digital artist) through doing (as a researcher) the same things that they do. In one sense, user-centred design has always valued the deep understanding of ethnographic participant observation. Indeed, the long-term research into artistic practices that informed the Palimpsest project had always been conducted in collaboration with academic anthropologists – initially as invited observers, then as project leaders and arts process researchers in their own right (Leach 2006, Barry et al 2008, Leach 2011).

However, this work stepped away from social science as a form of “requirements capture”, back to the traditions of craft professions in which tools are made and adapted by those who use them, learning with the hands and from the materials. The Palimpsest project intentionally blurred the boundary of software development and creative exploration, and deliberately avoided many of the conventional practices in each field. Rather than interdisciplinary collaboration and consultation, the central period of the research was conducted in isolation, living and working in a remote forest location in New Zealand, to provide a self-imposed focus on the personal experience of design.

As a meta-strategy for programming language research, this could be described as a process of “build, reflect, notate”, echoing the suggested end-user programming experience of “manipulate, automate compose” – but with the end result at this meta-level being a new notational convention for expressing and manipulating abstract computation. It is possible that this strategy may be particularly (or solely) suited to creating new programmable tools for use in artistic contexts. However the results of this project also suggest potential benefit of similar craft approaches in the design of end-user programmable and customisable products more generally.

5. References


The Lish: a data model to support analysis by end user programmers

Alan Hall
The Open University, UK
alan.hall@open.ac.uk

Abstract
For end user programmers needing to carry out data analysis, the spreadsheet is an attractive choice, but has little safety net against user errors. Reducing these errors is an active research area, but one aspect rather little investigated is the role played by the underlying data model: the grid of cells. I am working on an alternative model, the “lish”, based on nested lists of cells. Its theoretical advantages include fewer and more concise formulae, and easier updates to the structure. A user study is in preparation to assess its practical utility.

1. Introduction
The professional analyst needing to process tabular data has a number of options, including dedicated data science languages such as R, and libraries for general purpose languages, such as the SciPy library for Python. But for the end user seeking an interactive alternative, requiring little if any use of code, the most widespread choice is the spreadsheet. Its directness and immediacy make it accessible to non-programmers, but come at a price: spreadsheets are notorious for errors (EUSPRIG, 2018).

This situation presents a conundrum for researchers: it is tempting to try to augment the spreadsheet with features that might encourage users to adopt a more disciplined approach, but in so doing we risk losing the low barrier to entry and ease of use that made it so attractive in the first place. McCutchen et al (2016) take an alternative angle, arguing convincingly that a major flaw with the spreadsheet is that the grid of cells is simply the wrong structure for many of its use cases. It is broadly this line of attack that I pursue in my research.

If the grid of cells is the wrong structure, what is the right one? I am investigating the potential of a structure based on nested lists of cells, as the underlying data model for representing spreadsheet-like data. Like Miller & Hermans (2016) in their “gradual restructuring” work, I am seeking to allow users to capture additional structure where it would be helpful, while continuing to work in a spreadsheet-like environment. Another guiding principle is that of “tidy” data (Wickham, 2014). The new data model aims to be “tidy” on the inside, while giving the user the flexibility of layout they would associate with a spreadsheet.

2. Approach
2.1 The Lish data model
We have submitted a paper to the main conference track which gives more details of the model, called the “lish”. It is based on lists of cells, which may be nested so as to contain further lists; the first element of each list has a privileged status, forming a template which provides a minimum structure for subsequent elements.

The nested nature of the model has some obvious uses in capturing natural hierarchies within the data, and in permitting formulae to have multi-cellular results, whose size may not be known at design-time. The way the templates interact with the recursive structure causes some further useful properties to emerge. It enables systems of tables that have a repeating pattern to be guaranteed to repeat consistently. It can also avoid unnecessary formula replication, because the user can define a single formula in a template to apply to a range of other cells. In a normal spreadsheet, changing a value updates dependent values; in a lish, changing a structure updates dependent structures.

Templates are composed of ordinary cells and lists – they are not a separate abstraction. Hence they support the spreadsheet-like form of programming by example, where a user may define a calculation for a specific instance and then, by making it a template, expand it to a more general case.
2.2 Research questions
My top level question is “What are the pros and cons of expressing spreadsheet-like tabular data in a
nested list-of-cell form, as opposed to conventional grid form?” Some relevant sub-questions are:
“Does the nested form accord with users' mental model of their data” and “How does the nested form
affect users' workflow when conducting analysis”?

3. Current status
My initial work was to define the lish as a data structure and develop algorithms to operate on it.
Because a lish has more constraints than an ordinary list of lists, these algorithms have to
accommodate “action at a distance”, where modifying one part of the structure causes a corresponding
modification elsewhere. An example would be inserting a column in a table, which might cause a
separate column to be inserted in one or more related tables.

The second main piece of work was to define a “lish calculus”: a set of rules for performing
arithmetic and functional transformations upon lishes. This drew heavily upon the vectorised
arithmetic of the R programming language (R Core Team, 2018). The lish calculus allows calculations
involving many, possibly non-adjacent, cells to be defined using a single concise formula; the
structure as laid out in the templates is used by the machine to deduce which cells are to be operated
upon.

I have also built a small prototype editor, which allows the user to enter and edit a lish and build
formulae in a somewhat spreadsheet-like way.

4. Forthcoming work
Are the more “structured” data actually easier to work with? Are the more “concise” formulae actually
more intuitive to use? These questions can only be answered empirically, so my main outstanding
piece of work is a user study.

Participants will be government analysts, who are frequent spreadsheet users. They will be asked to
build a specified model in lish form, and this will be followed by a semi-structured interview. The
interview is intended to shed light on how well (or poorly) the nested form corresponds to users' mental model of their data. It will also elicit users' perceptions of the costs and benefits of structuring
data into lish form, with regard to their everyday workflow. And it will seek to identify any system
behaviour that was surprising to the user.

5. Conclusion
I have developed the “lish” on the hypothesis that it maps more closely to users' mental models of
their data than does a spreadsheet grid, and hence could facilitate building and maintaining end user
analyses with tabular data. The planned user study will help to assess whether these advantages can be
realised in practice.

References
EuSpRIG (2018) Spreadsheet mistakes - news stories. Available at: http://www.eusprig.org/horror-
stories.htm

for end-user development of data-centric web applications. Proceedings of the 2016 ACM
International Symposium on New Ideas, New Paradigms, and Reflections on Programming
and Software, 112–127.

Miller, G. and Hermans, F. (2016) Gradual structuring in the spreadsheet paradigm. Proceedings of
the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),
240-241.

R Core Team (2018) R Language Definition v 3.5.0, section 3.3, “Elementary arithmetic operations”.
Available at: https://cran.r-project.org/doc/manuals/r-release/R-lang.html

Modeling cognitive processes underlying computer programming

Cătălin F. Perticas
Romanian Institute of Science and Technology
perticas@rist.ro

Bipin Indurkhya
Romanian Institute of Science and Technology
and Jagiellonian University
indurkhya@rist.ro

Abstract
We present an approach to modeling computer programming as a cognitive process. In particular, we apply Piaget's four-stage model of cognitive development to study how programming is learnt by adult programmers. For this purpose we survey software developers at different stages in their career. In order to evaluate our approach, we analyze the gathered data through formal methods. Our approach is interdisciplinary in that it incorporates philosophical, psychological, cognitive and computer science aspects.

The results from this study will be used as a starting point for investigating the role of deeper cognitive processes underlying programming, which can offer hints to design improved neural architectures inspired from biology and cognitive science. Our vision is to use such models to generate programs automatically given the intention of the user.

The preliminary goal is to set connections between empirical evidence of how programmers write code, the cognitive processes implicated in software development and the corresponding mechanisms integrated in modern neural architectures. In the future, we plan to explore the potential of such enhanced neural models to solve tasks involving generation of computer programs.

1. Introduction
In recent years, there has been much interest in automatic generation of programs (Parisotto et al. (2016), Balog et al. (2017), Ling et al. (2016), Ling et al. (2017), Yin & Neubig (2017)). These computer programs are either induced - meaning that a neural network learns to behave like the desired program, or synthesized - the neural network is trained to output a program in a language of choice. We are interested in broadening these studies by designing and experimenting with computational agents capable of learning more diverse patterns of code.

Motivated by the past successes of neural architectures inspired from the human thought process and brain structures, we begin exploring cognitive mechanisms underlying programming, which can be modeled mathematically and ultimately implemented in a neural structure yielding a computational agent capable of writing useful code.

In the research described here, we are interested in modeling the cognitive processes underlying computer programming, and then using this model to automatically generate programs given the user intentions. We take our point of departure from our earlier work on relating Piaget's interaction view of cognition with software engineering (Indurkhya (2002); Indurkhya (2003)). We incorporate more recent research on applying Piaget’s theories to model programming (Corney et al. (2012); Lister (2011); Swidan & Hermans (2017); Teague & Lister (2014)). Then we take steps in the following direction:

- Research on how programming is viewed: paradigms, styles, conventions, abstractions
- Integration of research in survey design
- Collecting data for survey
- Modeling survey data for sub-groups and concepts analysis
- Investigation of extracted concepts from a cognitive perspective
- Implementation of cognitive processes inside a computational agent that generates programs
2. Philosophy of Programming

Over the years, programming has been viewed in a myriad of ways. For example, Graham (2003) argues against the traditional view that programming is a science, and relates the process of programming with the process of sketching or painting, thus linking coding or hacking to creative activities, rather than logico-deductive reasoning. (See also Hermans (2017)) Some of the pioneers of computer science, such as Edsger Dijkstra and Donald Knuth, have also subscribed to this view.

For instance, Dijkstra (1971) emphasizes the importance of good taste and style in programming by making the following analogy: teaching programming like a teacher of composition at conservatory - instead of teaching how to compose a particular symphony, help pupils find their own style. Similarly, Knuth (1968) titled his monograph, which laid the foundation of computer science, The Art of Computer Programming. He argued that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

Indurkhya (2002) noted, "Software is a rather unique entity. On one hand it can be considered a mathematical object — its component parts and operations of construction are rigorously defined, and the output result of a piece of software can be predicted precisely, at least in principle. On the other hand, it is also an empirical object — a piece of software executing on a machine is a physical object that can, as most of us must have experienced on many occasions, produce unexpected and unforeseen behavior. Moreover, as with any physically machinery, one can experimentally tinker with a piece of software and observe the consequences empirically."

Such is the case with machine learning, computational physics and other computational sciences, which involve programming for the purpose of simulating theoretically defined processes. The building blocks of these fields of study are mathematical models and simulation methods. For instance, a typical neural network is equipped with both a mathematical model represented by the structure of the neurons (layering, fully-connected, shared weights, recurrent and skip connections) and their operations (weighted sums, non-linear activations); as well as with a learning/optimization method (gradient descent, nearest neighbors).

Thus, neural networks, which combine symbolic software with numerical software, are programming entities which reside as objects in two separate spaces: the mathematical world - an architectural, biologically inspired object; and the empirical, experimental world implementing the dynamics of the interaction of complex systems, which to some extent replicate the human thought processes.

3. Psychological Factors in Style Formation

By interviewing different groups of programmers and non-programmers, we seek to gather evidence to support the view that our motivations, background and everyday activities shape the type of programmer, engineer or computer scientist we can become. Moreover, the tools we use, and the people we interact with, develop and influence our vision of a skilled programmer. Later on, if our training path is successful, we get to use these skills in new creative ways. However, getting to the creative stage requires, but is not guaranteed by years of experience, mentors with insights, and a healthy learning process.

In his book, Mastery, Greene (2012) presents the stories of creative geniuses from a similar perspective: how their background, motivations and mentors shaped their path to mastery. The apprenticeship model developed during the Renaissance period is of key importance here. There is a nice story - Zarnescu (2007) about C. Brancusi leaving the workshop of A. Rodin, both of them being very influential sculptors of the 19th century. Brancusi stated that "nothing grows under the shade of big trees". Their styles are obviously very different - as can be seen in Figure 1 - almost as if Brancusi was purposely trying to distance himself from the influence of Rodin.
A first glance at the two artworks reveals that Rodin put many details in his work, while Brancusi was an essentialist. Thus, Rodin’s work reveals intent, while Brancusi’s work implies it and leaves some room for interpretation. We can extrapolate this simple idea to programming.

It is very common for programmers to be fond of certain programming languages because they differ in style and expressiveness. For instance, someone who wants more control of variable types and syntax checks - a 'detailist' will prefer C++ or Java, while someone who prefers simplicity - an 'essentialist' will prefer Python or Ruby. Of course, the complexity of these programming languages goes beyond this trivial example, but our purpose is to argue that programming exhibits a style component.

Identifying more specific programming styles can be done by surveying across entire programming communities. It is interesting to notice that historically programming styles translated into programming paradigms. These paradigms evolve gradually inside programming communities and can lead to the development of new programming paradigms. From this point of view, software engineering exhibits both a cultural component, as well as an innovation component.

4. Cognitive Development through Stages

We aim to analyze the evolution of a programmer from a beginner to an expert in terms of Piagetian stages of cognitive development. In particular, our goal is to tease out changes in the thought processes of learners that allow them to make quantum improvements to progress to the next level. Towards this goal, we plan to conduct an empirical study based on the surveys of non-programmers and programmers at different levels (from novice to expert).

There have been similar studies in the past for incipient stages of learning to program (novice programmers) - Lister (2011), Corney et al. (2012), Teague & Lister (2014), Swidan & Hermans (2017). However, these studies have mainly focused on the educational aspect of programming (how to teach programming); whereas we are interested in studying developmental shifts which span longer time frames, from apprenticeship to mastery.

These developmental shifts reveal changes in how certain already existing cognitive processes are used in coding by experienced programmers, while they are not by novice programmers. Some of these processes can actually be learned quickly and prove very useful, but because they are not required until a later stage, their development is paused. However, the ones involving more effort to grasp, but are required for reaching a more immediate stage, are more heavily invested into.

4.1. Stage One: Simple Reflexes

Understanding the building blocks. It has been argued that mathematics is grounded in human activities (Mathematics, Form and Function, Mac Lane (1986); Where mathematics come from, Lakoff & Nunez (2000); The Number Sense, Dehaene (1996)). Piaget has also argued that mathematical concepts arise gradually from sensorimotor actions in Biology and Knowledge, Piaget (1971). Based on this, we hypothesize that how we write programs is highly influenced by our interests and by our simple interactions with the computer.
<table>
<thead>
<tr>
<th>Human Activity</th>
<th>Related Math Idea</th>
<th>Math Technique</th>
<th>Programming Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collecting</td>
<td>Object Collection</td>
<td>Set, class, multi-set, list, family</td>
<td>Array, list, objects and instances</td>
</tr>
<tr>
<td>Connecting</td>
<td>Cause and Effect</td>
<td>Ordered pair, relation, function, operation</td>
<td>Dictionary, Graph algorithms</td>
</tr>
<tr>
<td>Endless repetition</td>
<td>Infinity, Recursion</td>
<td>Recursive set, infinite set</td>
<td>For and while loops, recursion</td>
</tr>
</tbody>
</table>

*Figure 2 – An extension to connections drawn from Mathematics, Form and Function*

To test this hypothesis, we conduct short interviews of some recently 'self-made' programmers, who had different jobs before but decided to learn programming without any formal computer science education, and eventually managed to get jobs as software developers. We expect our analysis to reveal that what they were doing before drove their current interests in programming. Moreover, their concept of programming is influenced by the tasks they were doing before via analogies.

Figure 10 shows some information extracted from the interviews conducted. Most investigated cases in the 'self-made' sub-group revealed that programming is viewed as a tool - the means to achieve a goal. Side interests related to programming are either derived from their background or technology trends common in their work group. Their other side interests influence their interactions and role within their work-group, as well as their developmental vision - whether they are theoretically inclined, heading towards research directions, or practically inclined, thus preferring the engineering side of programming.

**4.2. Stage Two: Pre-Operational**

*Functional thinking.* According to Piaget, at this stage children do not yet understand concrete logic and cannot mentally manipulate information. Children’s interest in playing and pretending also takes place in this stage. However, the child still has trouble seeing things from different points of view. The children’s play is mainly categorized by symbolic play and manipulating symbols.

To study the cognitive processes underlying programming corresponding to this second stage, we choose to observe the behavior of a few summer interns at a research institute. Generally, internships are used by modern day companies to create proof of concepts for some ideas for which internal staff cannot be allocated. Interns are viewed as helpers, who are motivated by learning practical skills in a working environment, in a similar fashion to the apprenticeship model detailed by Greene (2012).

Given their motivation for learning and a lack of practical experience, we could consider their activities as playing in a work environment, which would correspond to Piaget’s pre-operational stage. The interns typically do not see their work from a business perspective, which is consistent with Piaget’s observation that children do not see things from a different perspective at this stage.

Figure 12 displays some of the attributes and tasks performed by the surveyed interns. We found that they managed to come up with good research ideas based on the topic they were provided with. Their imagination was quite rich and given proper guidance, they showed the ability to translate some of their ideas into actual software. However, they did not have so much success with the more complex topics they wanted to explore, either because they were not able to articulate their vision well, or because their technical expertise was not yet good enough.

At this stage, they still need someone to help select the most promising ideas out of the technically feasible ones. Another important aspect was their steadiness in solving the more challenging tasks. This seemed to be associated with both internal factors - how happy they were with their research topic and their work flows; as well as with external topics - the amount of encouragement they received for following their ideas.
4.3. Stage Three: Operational

During this stage, Piaget noted that a child’s way of thinking starts to be more adult-like. Problems are solved in a more logical fashion. However, abstract thinking is not yet developed so children can only solve problems that apply to concrete events or objects. Nonetheless they can generalize by making inferences from observations.

As a setting for the third stage of development, we chose to observe the environment of high-school programming competitions. The reason behind this choice is that good competitors are people who already know how to program, who have developed their problem-solving skills and implementation abilities, and who are able to generalize concepts across various types of problems.

At this stage, programmers can swiftly manipulate common patterns or templates used in programming competition problems. Moreover, they can reliably estimate the necessary time to implement a well-defined idea, as well as to precisely put their solution in practice. The book *Psychology of Coding Competitions*, Francu (1997) is particularly addressed to such high-school students. It suggests various strategies for training and problem solving.

The take-away message is that problem solving skills, which incorporate the ability to transform conceptual ideas into concrete algorithms, do not guarantee the success in competitions without a well-defined strategy belonging to the domains of psychology, decision making and time management. These concepts are mostly reflected in the thought processes of adult minds.

Towards the end of this stage, we can already observe the need for strategical thinking, which according to Piaget, is a cognitive process that is predominantly observed in the next developmental stage. Going back to the high-school programming competitions, we find that mentors play a key role in conveying strategical planning and decision making. Figure 13 shows focus areas and other features of various mentors we surveyed.

4.4. Stage Four: Formal Operational

Piaget’s theory states that abstract thinking, meta-cognition and problem solving are developed during this stage. In programming, these thinking patterns translate into a deeper understanding of programming concepts and how they relate to each other.

In the field of programming, the fourth stage of development is best represented by senior developers, who typically have more than 5 years of experience (industry standard). This stage coincides with the crystallization of the programmer’s application domain or specialization.

Senior programmers show a high degree of technical expertise in their specialization (in-depth knowledge). They have a broad view of other specializations (wide knowledge). And they are able to independently create fairly complex software or test research ideas end-to-end. Figure 11 displays information on surveyed senior programmers.

5. Formalizing Piagetian Attributes for Concept Analysis

The surveys we have conducted so far are insightful from a psychological point of view: we can observe trends in the developmental process of a programmer and how their background and motivations influence their learning curve and application domain.

However, different participant groups have been analyzed from different angles. At the same time, the data gathered is expressed in natural language. This makes it hard to make rigorous empirical observations that would generalize to new participants. For this reason, we started working on a set of principles to make such studies more generalizable and easier to interpret.

Our first principle is to convert a sample of Piaget’s developmental stages into the corresponding stages for learning to write programs. The second principle is to have a list of skills/thinking patterns specific to each developmental stage - Figures 3 and 4. These skills are converted into attributes, which can be subjectively quantified based on surveys - Table 5.
### Figure 3 – Summary of Piaget’s development stages from The Psychology Notes Headquarter - image source.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Age Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensorimotor</td>
<td>0-2 years</td>
<td>Coordination of senses with motor responses, sensory curiosity about the world; Language used for demands and cataloguing, Object permanence developed.</td>
</tr>
<tr>
<td>Preoperational</td>
<td>2-7 years</td>
<td>Symbolic thinking, use of proper syntax and grammar to express full concepts. Imagination and intuition are strong, but complex abstract thought still difficult. Conservation developed.</td>
</tr>
<tr>
<td>Concrete Operational</td>
<td>7-11 years</td>
<td>Concepts attached to concrete situations. Time, space, and quantity are understood and can be applied, but not as independent concepts.</td>
</tr>
</tbody>
</table>

### Figure 4 – Piaget’s stages applied to programmers (our model) and the description of stage-emergent attributes/thinking patterns.

<table>
<thead>
<tr>
<th>Id</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Coordination: Sense to Motor (CSM), Knowledge of Terms (KT)</td>
<td>The ability to implement straightforward ideas using a programming language. Knowledge about simple terminology, such as variables, lists, for loops, objects, classes, etc.</td>
</tr>
<tr>
<td>B</td>
<td>Symbolic Thinking (ST), Imagination (IMAG)</td>
<td>Understanding the purpose of programming constructs and their usage. The ability to use the knowledge about programming for devising new applications.</td>
</tr>
<tr>
<td>C</td>
<td>Coordination: Concept to Concrete (CCC), Knowledge of Concepts (KC)</td>
<td>The ability to implement a fairly complex idea described at a conceptual level. Knowledge about recurring concepts in programming and their application domains.</td>
</tr>
<tr>
<td>D</td>
<td>Abstract Thinking (AT), Metaphorical, connectionist and analytical thinking (META)</td>
<td>Understanding the purpose of programming concepts, the ability to abstract many concrete situations at a conceptual level. Debugging at different layers of abstraction, connect concepts to generate new ideas, applying concepts from one situation to another via metaphors.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>CSM</th>
<th>KT</th>
<th>ST</th>
<th>IMAG</th>
<th>CCC</th>
<th>KC</th>
<th>AT</th>
<th>META</th>
<th>YoE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>1.0</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>6</td>
</tr>
<tr>
<td>A.2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>A.3</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.25</td>
<td>0.5</td>
<td>0.5</td>
<td>0.25</td>
<td>0.0</td>
<td>3</td>
</tr>
<tr>
<td>B.1</td>
<td>0.75</td>
<td>0.5</td>
<td>0.5</td>
<td>0.25</td>
<td>0.25</td>
<td>0.0</td>
<td>0.0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>B.2</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>0.25</td>
<td>0.25</td>
<td>0.0</td>
<td>0.0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>B.3</td>
<td>0.75</td>
<td>1.0</td>
<td>0.75</td>
<td>0.5</td>
<td>0.75</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td>B.4</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>0.5</td>
<td>0.75</td>
<td>0.5</td>
<td>0.0</td>
<td>0.25</td>
<td>3</td>
</tr>
<tr>
<td>C.1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>12</td>
</tr>
<tr>
<td>C.2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C.3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>14</td>
</tr>
<tr>
<td>C.4</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 5 – Id: Programmer Ids, A’s represent interns at a programming/research job, with formal education in computer science, but little practical experience. B’s represent programmers with formal education in a different field, but practical software skills. C’s are senior programmers with formal education and practical skills. Next 8 columns are subjective quantifications of different skill levels according to our scheme for Piaget’s stages applied to programming. These are drawn from the conducted surveys. YoE represents the number of years they have been exposed to programming.

The third principle in analyzing this data is to use a mathematical tool for drawing conclusions from the data. Because our goal is to extract thinking patterns of programmers, and find out how these apply to various sub-groups of programmers, we model the data through a concept lattice taken from the theory of Formal Concept Analysis Ganter & Wille (1999) - Figure 6.
Our findings show that there is a strong agreement between Piaget’s theory, as we modeled it for programmers (attributes and skill levels), and their corresponding years of experience. We now get to the fourth and final principle of exploring this type of empirical surveys, which is to analyze this data through emergent concepts (6 in this case) - Table 7.

<table>
<thead>
<tr>
<th>Formal Concept Name</th>
<th>Subjects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensorimotor + Preoperational</td>
<td>100%</td>
<td>All have coordination, knowledge of terminology, symbolic thinking &amp; imagination.</td>
</tr>
<tr>
<td>Almost Operational Type A</td>
<td>A.3 &amp; below</td>
<td>Knowledge of concepts, but low conceptual to concrete - <strong>theoretically inclined.</strong></td>
</tr>
<tr>
<td>Almost Operational Type B</td>
<td>B.2 &amp; below</td>
<td>Conceptual to concrete, but low knowledge of concepts - <strong>practically inclined.</strong></td>
</tr>
<tr>
<td>Operational</td>
<td>B.4 &amp; below</td>
<td>Abstract and creative thinking not yet developed for programming.</td>
</tr>
<tr>
<td>Postoperational</td>
<td>B.3 &amp; below</td>
<td>B.3 has 1 year of programming experience and a strong mathematical background =&gt; <strong>abstract thinking</strong> developed.</td>
</tr>
<tr>
<td>Formal Operational</td>
<td>A.1, A.2 &amp; C’s</td>
<td>All these programmers have at least 4 years of experience.</td>
</tr>
</tbody>
</table>

Even though the number of participants and attributes are limited in our study, this principled method can be applied to more complex datasets for the discovery of human concepts in a formally defined context. The subjectivity in the evaluation of programming skills can be overcome via triangulation or objective evaluations, such as problem solving tests.
6. Cognitive Processes underlying Programming

By going into more depth, we can investigate the role of our cognitive processes in solving specific types of problems involving the design of algorithms, their implementation and general problem-solving in programming environments. Our brains have developed a number of complex cognitive mechanisms and systems for finding solutions to problems, some of these are inherent, natural properties of the brain, such as the ability to perceive and to attend to certain stimuli, while others are emergent and require developmental transitions, such as those exemplified within Piaget’s theory.

Out of these, we investigate the role of perception and attention. These seem to play an important role, not just in programming, but in daily human activities, as well as in computational agents designed to mimic the human mind. Variants of these mechanisms have been formalized and integrated into applications of neural networks.

6.1. Perception

For instance, perception at the level of neural networks has produced different kinds of architectures. These range from the traditional multi-layer perceptron (MLP), which do not assume any correlation between input units, to convolutional neural networks (CNN) mimicking the visual processing system by assuming spatial correlations; and recurrent neural networks (RNN) which exploit temporal correlation of input units.

![Figure 8 – Left: CNN Perception. Right: RNN Perception.](image)

It it interesting to notice how non-programmers view code as hieroglyphics. Even programmers find it hard to decipher code with obfuscated or badly named variable names. Although variable and function names do not influence the computational aspect of source code, they have a high impact on our understanding and perception of the code because they serve as anchors or starting points for creating internal representations of code. Only internally well represented code can allow for useful manipulations of source code.

![Figure 9 – A. Huxley's Cerebral Reducing Valve - image source](image)

Internal representations of the problem setting are key ingredients in general human problem solving. These internal representations are obtained through filtering and abstraction of innate perceptions. These reduction mechanisms are developed for maximizing our adaptive fit and to help with efficiency and clarity. However, because of their top-down modulation - sensory system is subordinated to conceptual thinking system - solutions that do not fall in the common known patterns can be found in the background of our minds, but they are eliminated. Huxley (1954) offers a nice perspective of this idea in his book *Doors to Perception*, arguing that our minds tend to reduce, rather than produce - Figure 9.
6.2. Attention

Top-down modulation in cognitive systems is usually referred to as attention, which is the next process we are interested to explore. According to Jones (1890), attention is "taking possession of the mind, in a clear and vivid form". However, more modern psychologists - Cherry (2018), claim that attention is both a highlighter, as well as the withdrawal from some things in order to deal more effectively with others.

Thus, attention is limited, selective and it is a core part of any cognitive system. This seems natural given the limited number of sensors present in cognitive systems. Attention is required in order for perception to be meaningful; attention guides perception. The interaction between these 2 systems generates the ability to select limited, but useful information from an unbounded noisy environment. While perception represents bottom-up information processing, attention is top-down modulation based on expectation.

Programming requires the ability to focus. Whether the object of focus is a line of code, an entire procedure or a project, being able to write or change code functionality would not be possible without attending to key components and places. For instance, changing one line of code might affect the desired effect of other lines of code, not necessarily in the proximity of the changed line. It is important to pay attention to the parts of code that are conceptually related to the changed area of code because in most cases it is impossible to have a perfect view of the whole.

Attention plays an important role in recurrent neural networks used for machine translation (NMT). For years it seemed hard to model long-term dependencies in recurrent neural models. The Long Short-Term Memory (LSTM) model was specifically designed to deal with this issue and saw great success. However, the issue still persisted for sequence-to-sequence problems. The attentional mechanisms described by Bahdanau et al. (2015) and Luong et al. (2015) significantly improved the state of the art in automatic translation of a sentence from one language into another.

Intuitively, the improvement comes from the learnt ability to attend to certain key phrases instead of the whole sentence. Because the neural architectures are of the encoder-decoder type, the information extracted by using the whole sentence maps to a conceptual representation of what the sentence means as a whole. Details such as the gender of a noun or verb tense are lost in this representation. On the other hand, attention allows to process a sentence one phrase at a time, thus ensuring that low-level information is correctly integrated in the translation.

Similar attention mechanisms have been tested on visual problems, such as the one presented by Xu et al. (2016) for generating image captions, or the one by Mnih et al. (2014) for localization and detection of handwritten digits. Neural networks implementing attention exhibit yet another advantage, which is to provide humans the ability to see what they see. For instance, attention weights will highlight the area in the image used for producing a given result.

7. Conclusions

We started this study by analyzing how programming is regarded and by establishing its similarities to arts and crafts. This revealed the fact that programming has many sides to it, which are shaped by the psychological factors influencing the person who creates programs. The development of programming skills and tendencies was then modeled using Piaget’s cognitive theory. We applied this model to data extracted from interviews of various groups of programmers and realized the importance of quantifiable attributes when testing the reliability of our model. For this reason, we gathered a sample of numerical attributes denoting a variety of skill levels in programming for different subjects. Cognitive patterns corresponding to Piaget’s theory were then discovered using formal context analysis. Finally, we mentioned some cognitive processes involved in programming, which could be included in our model to improve its generality. Our plan is to gather more data and use our concept discovery model to find recurring cognitive patterns involved in programming, which can be integrated in an automated code generator agent, the same way attention and perception were used to improve performance of neural networks.

8. Acknowledgement

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014-2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.
References
<table>
<thead>
<tr>
<th>Id</th>
<th>Current Job</th>
<th>Job Type</th>
<th>Prog Lang</th>
<th>Interests</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Machine Learning</td>
<td>Research</td>
<td>Python</td>
<td>Cognitive Sciences and AI</td>
</tr>
<tr>
<td>C.2</td>
<td>iOS Developer</td>
<td>Industry</td>
<td>Swift</td>
<td>Hacking and Cryptocurrency</td>
</tr>
<tr>
<td>C.3</td>
<td>Machine Learning</td>
<td>Research</td>
<td>JavaScript</td>
<td>Neural Networks</td>
</tr>
<tr>
<td>C.4</td>
<td>WordPress Developer</td>
<td>Freelancer</td>
<td>C#</td>
<td>Robotics and Arduino</td>
</tr>
</tbody>
</table>

*Figure 11 – Table illustrating different types of senior developers.*

<table>
<thead>
<tr>
<th>Id</th>
<th>Background</th>
<th>Current Job</th>
<th>Side Interests</th>
<th>Insights</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>design &amp; crafts music</td>
<td>front end &amp; web design</td>
<td>personal growth &amp; spirituality</td>
<td>Helping the team to learn new technologies &amp; looking for new ways to improve built products. Programming viewed as a tool mostly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>digital art &amp; synthesized music</td>
<td></td>
</tr>
<tr>
<td>B.2</td>
<td>social sciences</td>
<td>full stack web developer</td>
<td>politics &amp; law</td>
<td>Always in a search for new challenging tasks because it stimulates learning. Programming is great because it is a highly-demanded job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.3</td>
<td>mathematics</td>
<td>machine learning programmer</td>
<td>neuroscience &amp; artificial intelligence</td>
<td>Reading &amp; thinking about new research directions. Programming is great because you can test ideas quickly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.4</td>
<td>journalism</td>
<td>quality assurance tester</td>
<td>machine learning &amp; automatic testing</td>
<td>Reading about the potential of AI to disrupt &amp; innovate technologies. Programming is interesting because of the social effects it has.</td>
</tr>
</tbody>
</table>

*Figure 10 – Table with illustrating data from ‘self-made’ programmers, with no formal education, such as a university degree in computer science.*
### Table 1: Suggested Research Topics

<table>
<thead>
<tr>
<th>Id</th>
<th>Predisposition</th>
<th>Background</th>
<th>Suggested Research Topic</th>
<th>Articulation</th>
<th>Effectiveness</th>
<th>Steadiness</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Technical</td>
<td>Web Scraping</td>
<td>Automatic Web Scraping from DOM features</td>
<td>high</td>
<td>medium-high</td>
<td>high</td>
</tr>
<tr>
<td>A.2</td>
<td>Tech. Creative</td>
<td>Debugging</td>
<td>Automatic Bug Finding from ASTs</td>
<td>high</td>
<td>high</td>
<td>medium-high</td>
</tr>
<tr>
<td>A.3</td>
<td>Creative</td>
<td>Games</td>
<td>Multi-Agent Learning Behavior in Simple Games</td>
<td>medium</td>
<td>medium-high</td>
<td>medium</td>
</tr>
</tbody>
</table>

*Figure 12 – Table illustrating data from interns in their 2nd year of computer science program.*

### Table 2: Different Types of Computer Science Mentors

<table>
<thead>
<tr>
<th>Id</th>
<th>Focus Area</th>
<th>Background</th>
<th>Explored Topics</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.1</td>
<td>Learnability</td>
<td>High-school Teacher</td>
<td>Basic programming constructs and techniques, fundamental algorithms and the thinking behind them</td>
<td>Synthesized</td>
</tr>
<tr>
<td>H.2</td>
<td>Strategic Thinking and Problem Solving</td>
<td>Doctorand and Ex-Contestant</td>
<td>Advanced problem solving, decision making and strategy in programming competitions and psychological preparation</td>
<td>Structured</td>
</tr>
<tr>
<td>H.3</td>
<td>Creative Problem Solving, Simplification, Analogies and Approximations</td>
<td>Current Contestant</td>
<td>Hacking, unconventional programming, style of thinking</td>
<td>Unstructured</td>
</tr>
</tbody>
</table>

*Figure 13 – Table illustrating different types of computer science mentors.*
Abstract

How can we help users discover and learn spreadsheet features? In this work-in-progress report, we present findings from semi-structured interviews with seven participants, which asked questions on the themes of learning and adopting spreadsheet features. We find that feature adoption in spreadsheets is informal, opportunistic, and social. There appear to be three components to feature adoption: discovery, expertise acquisition, and attention investment. Users fall on a spectrum of intrinsic motivation for learning. We close with a few reflections on how features might be designed with adoption in mind.

1. Introduction

Large, complex applications such as Microsoft Excel are feature-rich, but for users, discovering and learning to use features is a challenge. Designers have several methods for alleviating this problem, e.g., callouts (‘look at the new features in this version!’), tooltips, help menus, and automatic suggestions. But how well do these really solve the problem? Poorly delivered suggestions can be viewed by users as intrusive and annoying. Principles for mixed-initiative user interfaces have been proposed (Horvitz, 1999), including ‘minimize the cost of poor guesses,’ ‘consider the status of a user’s attention,’ and ‘employ socially appropriate behaviours’ – but it is unclear what constitutes ‘socially appropriate.’

The ability of users to discover and learn features is key to the success of applications like Excel. Our experiences of speaking with users suggested that rather than benefiting from prompts within the application itself, users pick up spreadsheet tool expertise informally, opportunistically, and socially. This motivated us to investigate whether it is possible to design for social discovery and learning.

2. Method

We recruited a stratified group of 7 participants, 1 female, aged 35-65, using convenience sampling. Participants varied in spreadsheet expertise from minimal usage with only occasional formulas, to high expertise usage including the development of customised add-ins. Participants used spreadsheets for a wide variety of applications such as budget management, energy modelling, futures trading, climate change assessment, accounts reporting, and examination marks processing.

We conducted semi-structured interviews using contextual inquiry (Holtzblatt & Jones, 1993) at the participants’ workplaces. The interview was structured around the following questions:

- How did you learn to use spreadsheets?
- How do you learn new features and techniques?
- Can you give me an example of when you learnt a new feature or technique?

Our choice of the phrase ‘features and techniques’ was intentional; we are interested not only in features (e.g., charts, formulae, formatting, etc.), but also techniques for applying those features. By ‘techniques’ we refer to craft practices and design patterns (such as the conventions of layout and formatting associated with ‘good practice’).
3. Findings
3.1. Learning in Excel is informal, opportunistic, and social.
Learning is informal because it is not typically in a classroom setting or with formal learning materials such as textbooks, Excel help, and documentation.

[P4, small company accountant] I’m sure people within the accountancy company taught me. They probably showed me… I was a functionary, I was a temporary worker. So I sat down, and they said ‘this is what you need to do, input these figures’ etc etc. So I didn’t even necessary pay attention to what I was learning because I didn’t care about the job. The job was a means to an end. But of course when it comes to starting a theatre company, or what have you, or doing my own personal taxes, I realised I have a skill. I have a knowledge base. So in some ways it’s being told, in some ways it’s watching, and of course you ask questions. You get interested in how people do things. So you ask questions. But I never did a course in Excel. I’ve never taken a formal learning in Excel.

[P5, large company accountant] I don’t think there’s enough in the Excel documentation online. Looking through the forums you can find people who have similar scenarios to that which you’re looking for. And so I’ve tended to find that a bit more helpful. Sometimes in the forums people will include formulas that you can literally copy and paste.

[P6, professional accountant] To figure out a regular expression I would go and look at an O’Reilly book. I still have an O’Reilly book on regex. But my colleagues would just Google it. And that’s the experience that I see my children and others going through. It’s not one of a formalised reference book. When I learnt SAS, there was a bookshelf and you pulled these tomes, these Dickensian volumes off the thing and wade through the syntax. They don’t do this anymore, they just Google it, they copy the code and stick it in.

Learning is opportunistic because it happens as and when learning opportunities become available, either motivated by a problem which needs new expertise to be solved, or through observation/inheritance of others’ worksheets.

[P2, examination marking overseer] I know what I know how to use because it’s what I’ve needed. Effectively, I’ve learnt the functions as I’ve come across a need to use them, rather than just learning things abstractly.

[P6] I knew very little about Excel until I got into a finance course in my second year at University [...] We were studying the arbitrage pricing model, and I tried to implement that in Excel on a Macintosh 2SE. [...] There was this part of the APT (arbitrage pricing theory) that required a normal distribution and there was no function in Excel to do that. And given an obstacle, I then had to get around it, because it was a University deliverable. Given that obstacle, many hours were spent beating my head against the glass pane of that Mac trying to work out how to do it, and as a consequence, I learnt Excel.

Learning is social because unless the query can be very specifically formulated, users are more likely to seek help from others than from online resources. Users often learn from their co-workers, who see themselves as ‘helping’ and not ‘teaching.’ Seeking help from others allows users to learn in-context, because they don’t have to re-interpret any examples (e.g., from online tutorial material) for application in their own domain:

[P3, energy demand modeller] I didn’t have any experience with Excel, so I learnt Excel over a period of 4 years or so. Consultancies work on an apprenticeship model, so there’s usually someone more senior on the project, who will advise. [...] you can access advice from people in your team and other gurus around the company [...] I normally ended up on
analytical modules on projects because that was my background so I ended up using Excel quite a lot. People of that kind in my company sort of pride themselves on their ability to use Excel effectively and efficiently.

[P4] I remember there was this very clever guy who knew a lot about computers who was also part of the team, and he was always willing to help. This was almost pre-email, early 90s. Sometimes I went to them and said ‘can you look at this, I don’t know how to do this, can you give me a hand’ and often they’d just do it and show me. And I became quite friendly with this particular person. So I felt able to disturb him. In this tedious environment, you had to find ways to actually be engaging with people. So it kind of helped to do that.

[P5] I used to have a colleague years ago. He was a real whiz with spreadsheets. Occasionally he would point out ‘oh, you can simplify this by putting this in there’ and I guess subliminally you take these things in.

[P5] Maybe with certain formulas that I’m including that they haven’t come across before. A few people have asked me ‘how could I work this out?’ and so I might talk them through it. Some of these things I appreciate might seem fairly simplistic when you’ve done them a few times but other people haven’t done them before so you know, helping them to understand how to do that. And then if they don’t get it the first time, going back and helping them until they get it. If they can refer to something in prior spreadsheets, it makes it easier for them to understand rather than just having to start from scratch by themselves.

[P5] A lot of this I’ve picked up on the job. There’ll be terms or formulas that I won’t understand because they’re far advanced beyond my level, so I will have to ask my colleagues to explain things to me. And sometimes because it’s so much their day-to-day work and they find it simple to do, they can come across a little frustrated that I don’t get it. And they have to explain these things to me. I have to not be worried about asking because I’m not going to learn if I don’t ask.

[P6] So the first thing you teach someone coming into the company even with Excel is you teach them the [proprietary database] etiquette. ‘This is how we pull it out. This is how the sheets need to look in order to interface with [proprietary database] so that we can use these sheets and other people can use them’ and you would guide them. It’s almost an apprenticeship, somewhat medieval in its nature. And then people would start to work and then you’d guide them for two or three weeks, through what is deemed ‘good practice’. And that includes for example annotations, it includes how you break up the sheets, how you title and label. It can include showing people how to build up an audit trail in the sheet. How to use begin and ends, for example. So you’d coach them through that so they wouldn’t try to start and do something stupid like use individual cell references. You’d help them through that process in order to make this stuff more reusable and less brittle.

[P6] We don’t write things down. That’s a deficiency in the profession. We do try and build personality cults, and that’s tacit. There’s no handbook on personality cults. If you’re good at your job then people will emulate you. If you win their intellectual respect, then they will emulate you. If you’re good at your job and you’re explaining it and they’re seeing the return, they will simply adopt those practices. And a good boss will drive those practices quite deep if they’re successful.

Because of the social nature of learning, design patterns and idioms often percolate from influential groups or individuals. Users often learn about a feature, technique, or design pattern when they see it in a spreadsheet someone else has made, and therefore visible features percolate better:

[P3] So for example at some point there was introduced SUMIFS. I must have just seen that in a spreadsheet somewhere and thought ‘what the hell is that?’ and at some point come back to it and looked at it in help, and decided that was the right thing to do.
[P3] Array formulae was just this sort of mythical thing, it was just sort of this secret incantation that experts knew and you sort of heard of from time to time, until you finally went and talked to them and said ‘what the hell is this?’ For me, someone gave me a spreadsheet with a funny curly brackets in, and I inadvertently deleted certain cells, and thought ‘what the hell is this?’ and so, and then you go talk to someone, and they say ‘that’s array formulae’ and you look them up on the web. As far as I know there is no official documentation of the underlying algorithm by which Excel calculates array formulae. By trial and error you learn that some functions like IF behave a certain way in array formulae, but other functions like SUM behave a different way.

[P4] I may have seen on other spreadsheets, notes and comments being used, and thought ‘oh that’s a good idea’ but I realise that I need them. I use them as aide-memoires.

3.2. Adoption consists of discovery, expertise acquisition and attention investment.

There appear to be three distinct components to feature adoption. The feature must be discovered, the user must acquire enough expertise in order to use it correctly, and the user must be motivated enough (believe it gives them a significant enough reward) to invest their attention (Blackwell, 2002) into actually using it.

Moreover, users can be placed on a spectrum between two types: low intrinsic motivation and high intrinsic motivation for adoption. Many users have low intrinsic motivation to acquire additional tool expertise (Aghaee, Blackwell, Stillwell, & Kosinski, 2015). Excel’s flexibility allows them to ‘cope’ in many ways: for instance, users can manually type computed data values instead of learning how to write formulae, users can employ arithmetic primitives (e.g., ‘+’) rather than learn more general functions (e.g., \texttt{SUM}); users can merely lay out data in a tabular structure, rather than learning to use the more powerful, formal ‘table’ features; users can apply filters within tables and copy/paste the filtered view instead of learning to use pivot tables, and so on. We have observed that users’ intrinsic motivation interacts with the three components of feature adoption in a manner that is summarised in Table 1.

For instance, users with high intrinsic motivation have the ability to realise the need for a feature and formulate search queries that allow them to learn from online fora:

[P1, climate change modeller] I went to those kind of fora, and then largely trial and error. So trying it out and putting myself in the developers mindset here ‘surely this is how they would have done it’ and you try out something assuming that this is how they’ve done it and see if it works as it’s meant to work. And in most cases I’ve sort of sorted it out for me.

[P2] Usually by googling and finding out one of the forums which says ‘ah, you need to use function such and such’. There are lots of websites telling you how to do things on Excel. So I come up with what I want to do, and then you google it. So somebody says ‘ah you need to use this formula/function’, or ‘you need to do a macro’, or whatever, and then it’s you know, go off and make it work, basically.

[P2] Yes. I don’t find help in Excel particularly useful as a way of discovering a function. It’s very useful when you’ve worked out what function, to understand how it works. But I don’t find it particularly useful for finding the right function to use. I think it works more as a reference manual for how a function works.

[P3] For instance in programming there’s a concept of local variables, so certain variables have scope and that’s often useful. So I think ‘surely Excel has something like that’ then after some hunting you discover that you can make named ranges local to a worksheet as opposed to global and that helps in certain cases.
High intrinsic motivation users have a lower threshold for attention investment:

[P3] I believed that the spreadsheet would in the end be more robust because your data changes, output changes, things change, and if you do things more generally you’re a bit more robust to those changes. Also in some sense the job is not very interesting and it’s more interesting if you’re learning interesting things. And of course in the end there’ll be tasks that were impossible to solve unless you know the more advanced features.

Depending on motivation, users have different attitudes to the use of advanced spreadsheet features:

[P3] And my approach to Excel was always to... one was on any given task, to try and solve the task that was one step more general. So if you have a particular model to build or task to solve, you can do the very specific thing you were asked to do but almost inevitably, come tomorrow, you’ll be asked to do something slightly different. Or you can solve a slightly more general problem than the one you’re given which has the disadvantage that it takes a bit longer to get to the first answer, but has the advantage that it’s faster to get to the second answer when things change. In pushing oneself in that way, you learn to use Excel.

[P6] And a lot of my academic progress has yielded insights into that for example running macros and all the rest of it. Often in the workplace you don’t have a lot of time to do a lot of macros because the world evolves faster than the macros do. Other people may give you a different experience but automation to me is a very fine double-edged sword. You can achieve efficiencies but it is significant cost initially and if the world changes an inch, you can negate any investment. So a lot of my colleagues never automate beyond formulae, if that makes sense. And they’ll pivot table and they’ll refresh but they won’t do a lot of macros.

<table>
<thead>
<tr>
<th>Feature discovery</th>
<th>Low intrinsic motivation</th>
<th>High intrinsic motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive discovery</td>
<td>Users are autodidacts, have the ability to realise the need for a feature and formulate specific queries to learn about features from documentation, online fora.</td>
<td></td>
</tr>
<tr>
<td>Expertise acquisition</td>
<td>Informal, opportunistic and social. Learning occurs when a learning opportunity becomes available, but is not sought, and may not be connected to usage opportunities.</td>
<td>Learning is still usually informal and opportunistic, but is less likely to be social. Trial and error is a very common learning strategy. Usage opportunities create learning opportunities.</td>
</tr>
<tr>
<td>Attention investment</td>
<td>Users need strong evidence of reward from using a technique or feature.</td>
<td>Users have a lower threshold for evidence of reward. A bricoleur/technophile personality may push some users to apply new features even when there is no direct benefit.</td>
</tr>
</tbody>
</table>

Table 1 – Differences in adoption practices between users with low and high intrinsic motivation.
4. Design implications
What can the insights from this study suggest about designing for greater adoption?

1. Design for percolation: a feature whose presence is visible in a spreadsheet is more likely to be discovered than an invisible feature. This is not about the feature itself being visible - for instance, we are not advocating that the button for the new feature should be placed prominently in the interface. This is about the usage of the feature being apparent - it should be clear, if possible, that the feature has been used. This can sometimes lead to tensions in design. For instance, consider sheet-defined functions (Peyton Jones, Blackwell, & Burnett, 2003), which enable users to define new functions using formulae already present in the grid. Designers might want functions created in this way to look exactly like built-in functions so that users don’t need to think about whether they are using a user-defined function or a native Excel function. However, that would design against percolation of the sheet-defined function feature; making it invisible would reduce the propensity for users to opportunistically learn about it.

2. Design for explicit reward. For example, previous work has shown how it is possible to harness curiosity to incentivise users to write more complete tests in a spreadsheet-like end-user programming environment (Wilson et al., 2003). When designing new tools, consider whether it is apparent to the user how using it might reward them for their invested attention.

3. Include influencers as part of the user-centric design process. Microsoft’s MVP program1 shows how such individuals might be identified. If influencers are passionate about, and convinced about the utility of these new abilities, they are likely to be motivated to use them in their spreadsheets, and encourage others to use them too. These users are also likely to have excellent example use cases in which to ground the design process.

5. Related work
Small groups of people within organisations have been found to be responsible for sharing files, establishing and perpetuating ‘informally-defined norms of behaviour’ (Mackay, 1990). These people could be subdivided into two groups: (1) skilled, highly-motivated end-user programmers who were intent on experimenting and learning the software, and (2) less-skilled end-user programmers who were interested in interpreting needs of colleagues and creating files that solved those needs, facilitating communication between the technical group and the rest of the organisation.

Our data supports previous findings that while beginners learn spreadsheets mainly socially through colleagues, experts are more likely to further their knowledge using books, manuals and online resources, and in either case formal training is not common (Lawson, Baker, Powell, & Foster-Johnson, 2009; Nardi & Miller, 1990).

Our data suggested that the flexibility of spreadsheets permits a variety of ‘coping mechanisms’ for users to deal with low expertise, without having to acquire additional expertise. These coping mechanisms can be characterised as ‘bad practice’ (Lawson et al., 2009), which differentiates experts and non-experts; experts perform more planning and design activities when writing spreadsheets. A particularly interesting coping mechanism in highly collaborative environments is delegation: non-expert spreadsheet users collaborate with experts who can complete high-expertise tasks, therefore alleviating the need to learn, although this collaboration sometimes has an informal learning outcome for the non-expert (Nardi & Miller, 1990).

Our data also supports previous findings that spreadsheet learning tends to be goal-driven rather than structured (McGill & Dixon, 2001), an approach that could lead to lower quality spreadsheets as users do not acquire principles of design. Unsurprisingly, spreadsheet users are usually focused on understanding their problem domain, not programming, and therefore while they might become more familiar with

---

1https://mvp.microsoft.com/
their domain due to spreadsheet experience, they might not necessarily become better programmers with experience (Grossman, 2007). A related study of web designers found that the decision to learn is more often a matter of necessity than curiosity (Dorn & Guzdial, 2010). A think-aloud study of 10 participants (Reimann & Neubert, 2000) found that participants who self-explain while trying to learn how to use spreadsheets prove to be better problem solvers. Another study found that users learn spreadsheet skills more effectively through problem solving than through watching tutorials (Kerr & Payne, 1994).

General studies in programming expertise show that expertise in programming can manifest in a number of ways. For example, expert programmers have highly organised knowledge that allows them to do better at recall tasks (Wiedenbeck, 2005; Chi, 2006). Beginner programmers understand individual lines of code but not the relationships between them; experts see the more abstract, overall pattern of a program (Lister, Simon, Thompson, Whalley, & Prasad, 2006). Our data did not directly indicate that this was also true of spreadsheets, but would make for interesting future work.

Although we did not find evidence that users acquired expertise through learning scaffolds embedded in the spreadsheet packages themselves, previous work has found that tools that 'self-disclose' assist with learning end-user programming systems (DiGiano & Eisenberg, 1995; DiGiano, Kahn, Cypher, & Smith, 2001). Our observation that features whose use is visible in the spreadsheet creates learning opportunities suggests that scaffolding social interaction or stimulating information seeking behaviours might better integrate with users’ existing learning practices.

A longitudinal study showed evidence for gender differences in attitudes to technology adoption (Venkatesh, Morris, & Ackerman, 2000). Women were strongly influenced by the subjective norm (expectations of society to be ‘normal’) and perceived behavioural control (someone's perception of how well they are able to control some desired behaviour of their own – a sense of agency). A key aspect of this study was that there was strong top-down motivation (from managers) to adopt a new technology. This may be the case for some Excel users, but for many there is no such compulsion to adopt new Excel features. Gender differences may be less prevalent in users with high intrinsic motivation.

6. Conclusion

In an semi-structured interview conducted with seven participants, we have begun to learn how people learn spreadsheet packages such as Excel. The preliminary findings indicate that learning is informal, opportunistic and social, and that feature adoption in Excel consists of discovery, expertise acquisition, and attention investment. Users differ with respect to their intrinsic motivation to adopt spreadsheet features. In future work we intend to study whether these observations are also true of larger samples, as well as investigate more deeply how users learn specific features, such as formulae.

7. References


Abstract
Algorithmic music composition involves specifying music in such a way that it is non-deterministic on playback, leading to music which has the potential to be different each time it is played. Current systems for algorithmic music composition typically require the user to have considerable programming skill and may require formal knowledge of music. However, much of the potential user population are music producers and musicians (some professional, but many amateur) with little or no programming experience and few formal musical skills. To investigate how this gap between tools and potential users might be better bridged we designed Choosers, a prototype algorithmic programming system centred around a new abstraction (of the same name) designed to allow non-programmers access to algorithmic music composition methods. Choosers provides a graphical notation that allows structural elements of key importance in algorithmic composition (such as sequencing, choice, multi-choice, weighting, looping and nesting) to be foregrounded in the notation in a way that is accessible to non-programmers. In order to test design assumptions a Wizard of Oz study was conducted in which seven pairs of undergraduate Music Technology students used Choosers to carry out a range of rudimentary algorithmic composition tasks.

Feedback was gathered using the Programming Walkthrough method. All users were familiar with Digital Audio Workstations, and as a result they came with some relevant understanding, but also with some expectations that were not appropriate for algorithmic music work. Users were able to successfully make use of the mechanisms for choice, multi-choice, looping, and weighting after a brief training period. The ‘stop’ behaviour was not so easily understood and required additional input before users fully grasped it. Some users wanted an easier way to override algorithmic choices. These findings have been used to further refine the design of Choosers.

1 Introduction
Algorithmic composition typically involves structural elements such as indeterminism, parallelism, choice, multi-choice, recursion, weighting, and looping (Jacob, 1996). There are powerful existing tools, such as Max (Puckette, 1991) and SuperCollider (McCartney, 2002) for manipulating these and other elements of music. However, while these systems give great compositional power to musicians who are also skilled programmers (Wilson et al., 2011), many musicians who are not also expert programmers find these tools inaccessible and difficult to understand and use (Bullock et al., 2011).

This paper presents an evaluation of a prototype visual programming language (Bellingham et al., 2017) designed to allow structural elements of the kind involved in algorithmic music composition to be readily visualised and manipulated, while making little or no demand on programming ability. This system, called Choosers, centres around a novel non-standard programming abstraction (the Chooser) which controls indeterminism, parallelism, choice, multi-choice, recursion, weighting, and looping.

In this paper we present a programming walkthrough evaluation carried out with seven pairs of undergraduate Music Technology students. The purpose of this evaluation is to:

- Test the ability of self-taught music producers without programming skills to use Choosers to carry out a range of rudimentary algorithmic composition tasks;
- Identify usability and user experience problems in the current design;
- Identify tensions and trade-offs in the interaction design of the system.
In the evaluation, pairs of participants were introduced to each element of the graphical programming language via short tutorial videos. Participants were given a range of practical tasks to complete on paper or a whiteboard. The facilitator played a Wizard of Oz role, rapidly translating participants’ graphical solutions into runnable code that was fed into a non-graphical prototype version of Choosers so that participants could hear the musical results of their attempts.

2 Related work/problem setting

Various music programming languages are capable of algorithmic composition, although they require significant programming skills (Bullock et al., 2011) and are therefore inaccessible to many users. Bellingham et al. (2014) used the Cognitive Dimensions of Notations framework (Green and Petre, 1996) to review the usability of a representative selection of software capable of algorithmic music composition. The findings of the review included the following. First, we found that most existing software requires the user to have a considerable understanding of constructs in either graphical (e.g Max, Pure Data) or text-oriented (e.g. SuperCollider, ChucK, Csound) programming languages: such knowledge requires a significant learning overhead. Second, users are often required to have an understanding of musical notation and/or music production equipment such as mixing desks and patchbays. Third, several programs imposed working practices unconducive to compositional processes. Fourth, in some cases the user was unable to define, and subsequently change, the musical structure. Finally, complex visual design in graphical programming languages led to patches with multiple connections, making them difficult to read and to navigate.

3 Introduction to the system: Choosers

The following section provides a brief overview of Choosers, designed to cover enough detail to allow readers to understand the evaluation. Full details of the system design can be found in Bellingham et al. (2017). The system has general musical expressivity, but for simplicity the present evaluation focuses on the manipulation of samples for algorithmic composition.

Samples are shown in boxes, and can be auditioned by clicking on them. Samples can be assembled into sequences using arrows (see fig. 1). Samples in a sequence play in the order indicated by the direction of the arrows. Only a single arrow can enter or exit each element in a sequence. This deliberate limitation reflects the fact that parallelism and choice are dealt with elsewhere in the language. Boxes and sequences can be put inside other boxes, thereby packaging them into a single unit.

Boxes referring to samples or sequences can be snapped together vertically to create what are known as Choosers. Fig. 1 shows a Chooser with two lanes, each containing a sample (drums and bass). The number 1 in the nose cone indicates that at run time, just one of the lanes will be selected at random (subject to restrictions described below). By manipulating the number in the nose cone, any number of lanes from 0 to 2 can be chosen randomly to play simultaneously. A Chooser can have any number \( n \) of lanes. By manipulating the number in the nose cone, any number of lanes from 0 to \( n \) can be chosen randomly at run time and played simultaneously. Each lane has a weight associated with it. Consequently,
in fig. 1, the drums are twice as likely to be chosen as the bass. Additionally, a weight of ‘A’ (‘always play’) can be used to ensure that the lane is always selected for playback.

Any sample can be set to loop indefinitely when selected on a particular run, or to play just once by the choice indicated in the status column (shown in fig. 1): indefinite looping of a single sample is typically not desired, so we now introduce **Time Choosers** (see fig. 2, left).

![Figure 2: An annotated Time Chooser (left); a Full Chooser (right)](image)

If the Time Chooser (fig. 2, left) is attached to the bottom of the Chooser (fig. 1, right) this produces a **Full Chooser** (fig. 2, right). When the Full Chooser shown in fig. 2 is played, looped drums, if chosen, cannot play indefinitely, but will be cut off after 16 bars. However, if the status column in the time chooser were set to > (indicating a soft stop) rather than × (indicating a hard stop) then, after 16 bars, the sample would play to the end of its current iteration. With a hard stop, if the Time Chooser duration cleanly divides the sample duration, every repetition will play in full. If not (e.g. if the bass.wav sample in fig. 2 had a duration of 3 bars) a hard stop will cut playback mid-sample. The two kinds of stop work similarly with non-looped lanes. If the non-looped bass lane of the Full Chooser (fig. 2, right) were chosen, the bass sample would be guaranteed to start playing once. With either kind of stop, if the sample were less than 16 bars long, there would be silence after completion until the end of the 16 bars. A non-looped sample longer than 16 bars would be truncated by a hard stop but allowed to complete by a soft stop.

Now that Time Choosers and Full Choosers have been introduced, in order to avoid ambiguity, we will refer to Choosers with no attached Time Choosers, such as those shown in fig. 1, as **Soundable Choosers**.

A Time Chooser can be used alone as part of a sequence – however, when used in this way it will simply result in a rest of the specified duration. More generally, the purpose of a Time Chooser within a Full Chooser is to moderate in a non-deterministic manner how long the Soundable Chooser and its individual lanes play. Possible interactions between the settings of soundable and Time Choosers can make the results more varied than might be imagined. A Time Chooser’s nose cone can be set to either one or zero. If set to one, one time lane will be chosen at run time. If it is set to zero no time lanes will be selected and the Soundable Chooser will run as though there is no Time Chooser. This allows for quick low viscosity (Green and Petre, 1996) arrangement changes, with the possibility of infinite playback if the Soundable Chooser lanes are set to loop. With no Time Chooser and the chosen lanes not set to loop, the samples will play and the Chooser will be released when they have finished playing, regardless of length.

### 4 Method

#### 4.1 Participants

Seven pairs of undergraduate Music Technology students took part in user tests utilising a Wizard of Oz prototyping methodology. These users were targeted as they are typically lack programming skill and extensive formal music training. While they may be conversant with some elements of music theory, the predominant background is self-taught music producers with experience of making music electronically using Digital Audio Workstations (DAWs).
All participants were asked to complete a short questionnaire before taking part in the user tests. Of the fourteen participants all considered themselves musicians. Six participants did not read any music notation (though ten had some formal musical training). Most of the music readers could read common music notation as well as chord notation. All participants were familiar with DAWs, with Logic Pro (Apple Inc., 2013) mentioned by all fourteen users. Other DAWs mentioned included Pro Tools (6 mentions), Cubase (2 mentions), FL Studio (5 mentions), Reason (1 mention), and Ableton Live (1 mention). Pure Data (Puckette, 1997) a visual audio programming language, was mentioned by two participants. Twelve participants had experience using hardware for music performance HCI tasks (such as drum pads or control surfaces). The participants were not habitual performers; seven of the fourteen participants do not perform with or for others. Of those that do, three perform in church, and four occasionally play with friends in private. Five of the fourteen participants claimed some experience in computer programming; however, two of these considered markup for the web (HTML and CSS) as programming. If simple markup and layout are excluded, then more than two thirds of participants (11 out of 14) lack experience of writing algorithms in any programming language. One participant listed the use of Pure Data and SuperCollider. Eight of the fourteen participants did not know what algorithmic music was at the start of the user tests. The remaining six participants felt that they knew what algorithmic music was but had not created any.

4.2 Walkthrough protocol
Participants were asked to take part in eight scenarios, as reproduced below. The users were free to discuss the work and to ask for clarification with the administrator of the test. Users were asked to act as active participants in the research, and to help in categorising any issues that were raised. The categorisations that users were asked to use – taken from the programming walkthrough method (Bell et al., 1992, 1991) – were questions (e.g. why does the loop do that?), problems (e.g. I don’t understand what these lanes are for), suggestions (e.g. maybe the cone should be a different shape), and other observations (e.g. I like the fins). In addition, participants were asked if they could think of any other ways in which each scenario could be completed. This prompted a discussion on alternative routes in order to test understanding and to capture user expectations.

4.3 Walkthrough scenarios
The eight scenarios issued as part of the user tests are shown in fig. 3. The users were introduced to each element of the graphical programming language via short tutorial videos1. Users were given a range of practical tasks to complete on paper or on a whiteboard (see fig. 4), and their outputs were played by the facilitator using a set of SuperCollider (McCartney, 2002) classes written to implement the musical abstractions behind the system. The user tests were videoed and transcribed to assist in the analysis presented here.

---

1 Available at https://goo.gl/PFeAJf
Figure 3: The eight user test scenarios
4.4 Results
All participants were able to understand the behaviour of Soundable Choosers. Twice participants had to be corrected after believing vertically stacked lanes all play concurrently – both instances of this misunderstanding occurred in the first scenario only. Two groups found multiple vertically aligned time lanes confusing and assumed that both durations would play together, or one would play before the other. The initial introduction of a Time Chooser as a representing a rest, and only later showing it constraining a Soundable Chooser’s duration, was confusing for four of the seven groups. Hard and soft stops were understood, but seven of the fourteen participants asked for clarification of the behaviour of the soft stop. Two pairs of participants suggested alternatives for the hard and soft stop icons. Two participants wanted to use ‘always play’ to express infinite playback rather than removing the Time Chooser. There was some confusion over the meaning of ‘always play’ and whether it could be skipped. Some felt the interface used numbers for too many parameters.

Three groups commented on the ‘boring’ design. The layout of Choosers was not seen as problematic, but some users wished for a more stylish and polished presentation. Two groups requested colours to enhance usability: within one group, one user wanted automatic colour selection (denoting lane type) and the other user felt that user-controlled colour selection would better support sorting and arrangement tasks. Several users were interested to know if lanes could be rearranged to visually organise lanes into instrument groups. One user suggested that lane arrangement could be an alternative to the weight column — moving a lane higher would result in a higher probability of playback. This is similar to one mechanism which was considered and rejected before the user tests: it was replaced by the weight column as the column allows for multiple identical lane weights, quick auditioning, and user-controlled lane ordering to assist with musical arrangement. One user requested instrument icons for lanes, partly in response to being unaware of the marimba (one of the samples used in the user test).

Overall, participants were able to complete all scenario tasks, with varying levels of assistance.

5 Reflection on design issues
The findings from the user tests outlined above have various implications for the design of Choosers.

5.1 Musical issues
Repeating phrases, and the musical interaction between phrases, are crucially important in a music system. These have therefore been brought to the surface via the loop and hard/soft stop behaviours. We found that the stop behaviour was confusing to four of the seven pairs of users, and the documentation will be enhanced to better explain the system. The hard and soft stop system can be conceptualised in a
number of ways. For musicians, one useful way is to consider soft stops as suitable for melodies, and hard stops for accompaniment. Melodies are therefore allowed to finish, whereas accompanying elements are stopped when the duration of the Chooser elapses.

Three of the fourteen users were keen to have a visual indication of current position with respect to duration, such as a progress bar. While this seems a reasonable request, it is complicated by the non-deterministic nature of the system.

A significant number of participants found the use of Time Choosers for both rests and Chooser duration to be confusing. This was largely due to rests being introduced before the Time Chooser’s primary function, which is to control the duration of a Chooser.

None of the participants had experience in algorithmic composition, so these sessions essentially introduced algorithmic compositional tools while also testing the interface. This led some participants to presume that the concepts themselves were novel. Some time was spent discussing the desirability of algorithmic processes rather than this specific implementation. Two participants assumed that the process would lead to a linear audio file, which indeed it can, but many use cases would require the music to remain nonlinear. Future evaluations could explore the nature of attitudes to nonlinear playback, including how it is related to expectations set by commercial music creation software and linear playback. We are also interested in the use of Choosers in genres which routinely incorporate extemporaneous changes and improvisation, such as folk and jazz.

One group specifically wanted a mechanism to allow them to easily reuse material for thematic development. The design of Choosers allows for this via the nesting of Choosers within lanes, although it was not included in the user tests for simplicity. The users were shown nesting in response to their questions and found it to meet the need they had expressed.

Choosers can be used in the creation of a range of music. However, given the unusual combination of usability and affordances, Choosers are particularly suited to music in which users would benefit from easy access to non-linear playback. Some classic Minimalism techniques (Potter, 2002), such as phasing (Scherzinger, 2005), are easily achievable using Choosers. Game music is often non-linear, created using layers of musical material which are triggered by in-game events (Collins, 2008). Such material can be created using Choosers, and we have a mechanism which would allow for external input via OSC or an alternative protocol; this would allow a game engine to trigger changes in the music. Choosers also allow musicians and music producers to create nonlinear versions of existing recordings by loading alternate takes into Choosers. The playback could range from very close to the original (e.g. algorithmically switching between vocal takes of the same melody) to playing significantly different material (e.g. branching to play different sections), depending on the decisions made by the creators.

5.2 Programming-related issues

As shown in sec. 3, the Soundable Chooser nose cone slopes down and the Time Chooser nose cone slopes up – this allows them to be joined together and communicates the required upper/lower order to the user. Interestingly, some users guessed the combination of Soundable and Time Choosers, suggesting that the nose cone shapes of the two Chooser types were effective in communicating their combinatorial usage.

The Chooser system is designed to allow for consistent logic to be applied across Soundable and Time Choosers where possible. Participants in the user tests successfully reused elements of the Soundable Chooser system when manipulating duration, but there were some cases where such reuse or re-contextualisation was not possible. Interestingly, the actions of the users in these cases would have made sense neither from a musical nor programming perspective, but the rationale behind these requests is instructive as it shows how users understand the tools in the system. For example, in scenario five (fig. 3) two participants wanted to use the ‘A’ (always play) mechanism to set infinite playback – they wanted to override the set duration and had understood ‘A’ to be a global override control. In a similar example, one
user wanted to be able to loop a Time Chooser. If the system were to be changed to allow for a set number of repeats, rather than an infinite loop, such a move may be desirable.

Users will also require access to metadata – for example, to check the length of a sample loaded into a soundable lane in a Chooser. Such metadata could be shown via a tooltip, accessed by hovering the mouse over a lane.

5.3 Shared and existing knowledge
One design motivation is to enable people to understand the system very quickly. The Chooser design tacitly draws on a number of systems of existing knowledge.

Some users wanted to be able to leverage their existing understanding of DAW software and found it frustrating that they needed to learn new paradigms for duration, synchronicity, and so on. This is an example of technological framing (Orlikowski and Gash, 1994). The knowledge gained by using other music software can be useful, but it can also prove problematic if the design of the software being learned is sufficiently differentiated. As a result, there is much to be gained by following standard design conventions where possible, as this maximises the user’s ability to reuse existing knowledge. One interesting example was seen in scenario 6 (fig. 3), in which one pair of users learned the rules of Choosers and then wanted to use the same rules elsewhere.

Technological framing, and the expectations set by the use of commercial DAWs, may be an influence on user requests for a progress bar and the conversations on the desirability of nonlinear vs. linear playback that took place during the user tests (as considered in sec. 5.1).

5.4 Metaphor
Interface metaphors are very common and can be useful in communicating the roles of the software and setting realistic expectations when users are familiar with the original interface. However, such metaphors can become problematic if users are not familiar with the original interface.

Related to technological framing is the assumption, ubiquitous in Digital Audio Workstations, that signal flow and processing will be applied using a mixing desk metaphor. Such virtual desks often make use of skeuomorphism (such as the fader caps and rotary potentiometers in Pro Tools), although some other designs have made graphical changes while retaining the overall layout. As an example, Ardour’s use of a textured ‘strip’ instead of a fader is still skeuomorphic as it makes use of a ribbon controller metaphor but, in an attempt to improve mouse control by increasing the size of the target, it does not follow the traditional desk layout.

Given that DAWs are now capable of performing all mixdown tasks, and the financial cost of consoles and outboard effects processors can be prohibitive, many users learn in a virtual studio environment rather than on hardware. Many DAWs were designed to mimic hardware in order to leverage existing knowledge and ease the transition from hardware to software. However, now most people are introduced to music production via software, and many do not use hardware, there is an opportunity to revisit some design assumptions.

Some users felt that the user interface was ‘boring’, lacking the use of colour, metaphor, and skeuomorphic design common in DAWs. This may be another example of technological framing (Orlikowski and Gash, 1994). We can also consider the impact of metaphor in music software by making use of the Cognitive Dimensions of Notations (Green and Petre, 1996). Using this framework, the closeness of mapping and role expressivity of a mixing desk can be implied by making a software recreation look and function like hardware.

Some users had difficulty understanding the outcome of hard and soft stops in Choosers. The vast majority of music production software is focussed on the creation of linear music, and the concept of ‘play until finished’ is rarely implemented. As a result, none of the user test participants had encountered it, and did not have a frame of reference for why it might be desirable. As a result, there is not a clear
existing metaphor for what we refer to here as a ‘soft’ stop. Users agreed that the × icon represented a traffic stop sign and that it was a suitable analogy for ‘stop now’, but the > icon used for a soft stop was not immediately understood as there is no readily accessible metaphor.

5.5 Arithmetic
The use of numbers and arithmetic relationships in an interface can be a valuable organising tool, as they are more or less universally familiar and can concisely represent many relationships. The decision to use numbers for several parameters was motivated by parsimony and consistency. However, the use of numbers for multiple parameters was perceived as negative by three participants. Upon questioning, the issue was that numbers meant different things in different parts of the interface. The Chooser design presented to participants in the user tests made use of integers in five different ways: for the number of simultaneously playing soundable elements, weight, duration, repeats, and Time Chooser on/off. Despite this, for different reasons, the user issues surrounding the ‘always play’ option led us to consider extending the range of numerical concepts used in the interface, by allowing the metaphorical use of ∞ as a weight (to outrank any positive integer weight) as discussed in the next section.

In sec. 6, we propose changes to Chooser design to address these various issues.

6 Design problems and candidate solutions in Choosers
Given the problems for some users with the use of integers for multiple parameters (see sec. 5.5), we propose the use of a simple on/off icon for the Time Chooser nose cone. Interestingly, one pair of users suggested this change in the user tests. Scenario five (fig. 3) showed that two users wanted to leverage the ‘always play’ mechanism beyond the weight column, and one user wanted to set the duration of a time lane to infinity. We propose a change to Choosers which allows for both mechanisms.

We propose a design change which allows the user to allocate a maximum possible weight (∞) for a lane, thereby guaranteeing that it will play if the nose cone number is high enough to allow all such lanes to play. When allowing ∞ as a weight, a useful metaphor is to think of lanes with weight ∞ as having paid for ‘priority boarding’, as when boarding an aircraft. Lanes with weight ∞ will always be chosen before any lanes with any finite weight. Compared with the ‘always play’ mechanism, this has the potential for greater clarity when the number of maximally-weighted lanes exceeds the number of the nose cone. In such cases, under the current ‘always play’ system it is not obvious whether ‘always play’ should override the nose cone or vice versa. Under the proposed system, the nose cone would determine the number of lanes to play, and if that was less than the number of lanes with weight ∞, the winners would be chosen from those lanes at random. We are also considering the use of a maximum value (∞) for the nose cone of a Soundable Chooser (‘play all available lanes’) and for the duration of a time lane (‘play forever’).

We propose that future work will introduce Time Choosers in the context of a Full Chooser, with the rest functionality introduced later as a special case. Tutorial materials will provide a clear explanation and will offer context and examples. The value of all of these proposed changes will be tested empirically.

7 Conclusions
Choosers were developed to allow non-programmers access to algorithmic composition tools and processes. The design principles were to leverage parsimony in order to enhance learnability; to surface musically meaningful actions, and to make them quick and easy; to allow both bottom-up and top-down construction; and to make use of progressive disclosure to allow for advanced use without harming usability for beginners.

The user tests outlined here show that non-programmers were able to successfully use Choosers to create a number of short pieces of music. Future work will focus on the refinement and re-evaluation of the Chooser notation and supporting materials.
References


All recordable human discourse is trapped in aleph-zero

Oli Sharpe
Go Meta
Brighton, UK
oli@gometa.co.uk

Abstract

Borges’ short story “The Library of Babel” is a classic literary exploration of the idea of a combinatorial library that contains all possible books of a certain format. However, this idea can be expanded into a theoretical enumeration over all possible recordable attempts by humans to communicate. From this we can deduce that the fidelity with which humans can refer is ‘only’ countably infinite, which is the smallest infinity known as aleph-zero. This paper constructs this enumeration and explores two consequences of it. Firstly, it is at least possible that the size of the set of ‘true’ things in the universe is a ‘larger’ infinite, such as aleph-one (as suggested by the diagonal arguments by Cantor, Gödel, Turing). If this is the case, then it would be impossible for even the full extent of our theoretically possible recordable discourse to explicitly refer to each thing that is ‘true’ about the universe. Maybe there are unquantifiable and indefinable aspects of the world that we cannot capture in any recordable discourse, let alone in the specialised discourse of programming. However, such an expressibility gap would be between humans and the universe, not humans and computers. Secondly, this paper looks at how this enumeration gives theoretical support for certain uses of unique identifiers in programming languages, such as Semprola.

1. Introduction

In this paper we are going to look at a particular limit on the fidelity with which humans can communicate. The reason we are interested in communication, rather than thought, is because the long-term value to humanity of a thought depends on our ability to communicate the thought to others.

And the reason for looking at this communication fidelity limit is to compare it to both the possible ‘fidelity’ of the universe and to the similar limit on the fidelity of what computers could communicate.

To examine this limit we will be building enumerations (complete, ordered listings) over sets by constructing indexes that give each member of the sets we’re looking at a unique identity, a ‘UID’ within that set. The last part of the paper will then link the discussion about the limits of communication with the use of UIDs within programming environments.

2. Cataloguing Communication

One way to think about a book is as an attempt by the author to communicate an idea to all potential readers of that book. A library is therefore an attempt to collate together a fairly comprehensive catalogue of all such attempts to communicate using books. But how comprehensive could this library be?

2.1 The Library of Babel

In ”The Library of Babel” Borges (1941) imagines a universe constructed as a single library with an indefinite number of identical looking hexagonal galleries holding books. These hexagonal galleries are stacked upwards and downwards as far as the eye can see with a shaft of air between all the floors and only a balcony to walk around to reach the books on each level. Four of the walls hold the shelves of books while the other two1 walls have passageways through to yet more of these hexagonal galleries. A door on each such passageway reveals a staircase to reach the upper or lower levels.

1 In the text Borges actually says only one wall has a passageway, but many readers assume that this is a slight editing error from what Borges intended, in particular see: https://libraryofbabel.info/theory.html
Each hexagonal gallery contains 700 books arranged and constructed as follows:

“There are five shelves for each of the hexagon’s walls; each shelf contains thirty-five books of uniform format; each book is of four hundred and ten pages; each page, of forty lines, each line, of some eighty letters which are black in color.”

The character set used for every book includes twenty-two^2 letters of the alphabet and the space, comma and full stop characters. The library contains every possible 410 page combination of these twenty-five characters. Given that blank pages can be constructed entirely out of the space character, this means that every piece of text of length up to 410 pages is held somewhere in this library, including this paper (albeit with a simplified character set). Indeed, in the Appendix below there are several examples where the abstract for this paper has been found in random books in an online version of the Library of Babel. Somewhere on its shelves the library also contains all drafts and reviews and rebuttals of this paper as well.

By imagining the combinatorial library as a physical space with people attempting to make sense of the books they find as they wander around this seemingly endless universe of books, Borges manages to convey the complete lack of utility of such an exhaustively comprehensive library. Picking up and reading a random book from the library is essentially pointless. Most of these random books are unintelligeable, and even those that can be read have no real meaning as they have no author attempting to directly or indirectly communicate an idea. Without an author there is no intention behind any of the text that happens to appear on the page, even if the text has the appearance of conveying an idea. We know that for any such apparent idea in any of the random books, there will exist another book in the library with the exact opposite idea, or with a convincing counter argument to the first idea. There is no communication going on, all these books just happen to exist.

Borges’ short story therefore makes clear just how fundamentally important the history of a given text is. For any text to be a genuine attempt to communicate a meaning there must be the intentions of an author behind the text.

Note that the ‘author’ of the text may have generated the text via a process of some kind (e.g. recording the temperature every hour), but the process must be of the kind where the semantic intention of the author is sufficiently preserved to ensure that the text itself has meaningful content. So, while the automated generation of random books may be achieved by a process created by an author with an intention to communicate some general idea about libraries of random books, the text of each particular random book thereby generated does not individually convey this general idea.

And of course there is a more complex potential example where the human ‘author’ has created a sophisticated artificial intelligence (AI) which in turn is generating text in order to convey ideas that the AI ‘intends’ to communicate. In this case we’d take the AI to be the author of the text and thereby see this as a very different case from the situation where a random text generator happens to have created text that appears to have meaning.

As all of the books in Borges’ library are randomly generated therefore they are devoid of specific semantic content and so it completely fails to perform the function of a real library.

2.2 Creating an index for an enumeration

None the less, the conception of Borges’ library gives a nice example of how a comprehensive set of things can be constructed combinatorially thereby making it easy to conceive of an enumeration that exhaustively covers the set.

In the case of the Borges’ library we can first give each of the 25 characters in the used character set a number between 0 and 24, and then give each character position in a book a unique number p where p = page x row x character in row, (with these positioning dimensions ranging from 0 to 409, and 0 to

^2 Borges uses a reduced character set for the construction of his library as explained in his essay, “The Total Library”. For more information about this see: https://libraryofbabel.info/theory3.html
39 and 0 to 79 respectively), so that each book is considered to be simply a string of 1,312,000 characters. Now we can calculate the unique index ID number for each book in the library as being:

\[
\text{Library of Babel unique ID number (LBUID)} = \sum_{p=0}^{1,312,000} c_p \cdot 25^p
\]

Where \(c_p\) is the number for the character in the \(p\)th position within the given book. Note that this number can not only be used as a catalogue number for the given book, but also contained ‘within’ the single number is an encoding of the entire content of the book itself.

One thing that this formulated enumeration makes abundantly clear is that Borges’ library is in fact not infinite in size\(^3\). There is a largest book index number.

However, in the next section we will look at how we can extend this notion of a combinatorially constructed ‘library’ to include an infinite number of books (strings) of arbitrary length and then extend it yet further to include all forms of recordable communication.

But, before that, let’s briefly look at the different kinds of infinity that are referred to as aleph-zero and aleph-one.

2.3 What is aleph-zero?
The set of natural numbers, \(\mathbb{N} = \{0, 1, 2, 3, \ldots\}\), has no highest number and so if you kept counting these numbers you would continue ‘forever’. The size of this set is therefore not finite, but infinite.

Mathematicians have noticed that there is a whole class of sets that can be mapped one-to-one onto this set of natural numbers, so we can know that these other sets are no bigger than and no smaller than the size of \(\mathbb{N}\). In other words, these sets are the same size of infinite. An important example of this is the set of rational numbers, \(\mathbb{Q}\). The one-to-one mappings are usually achieved by creating an enumeration over the set providing a unique index number (in \(\mathbb{N}\)) for each member of the set whose size we wish to compare to \(\mathbb{N}\). This is why we created an enumeration over the books in Borges’ library and will continue to look at enumerations in later sections of this paper.

In 1891 Georg Cantor published his diagonalization proof (Cantor, 1891) that shows there can be no such enumeration from the set of real numbers, \(\mathbb{R}\), to the set \(\mathbb{N}\). The set \(\mathbb{R}\) contains more things than the set \(\mathbb{N}\) and therefore the infinitely large size of \(\mathbb{R}\) must be strictly larger than the infinite size of \(\mathbb{N}\). The suggestion from this is that there are different ‘sizes’ of infinity!

Aleph-zero, \(\aleph_0\), is the name of the ‘smallest’ infinity used for countably large sets such as \(\mathbb{N}\).

Aleph-one, \(\aleph_1\), is the name for the next largest infinity and there are larger infinities like \(\aleph_2\) and so on. Cantor showed that the size of the set \(\mathbb{R}\) (denoted \(|\mathbb{R}|\)) was \(2^{\aleph_0}\) and proposed that there are no sizes of infinity between \(2^{\aleph_0}\) and \(\aleph_0\), which is equivalent to suggesting that \(\aleph_1 = 2^{\aleph_0}\). This is known as the continuum hypothesis and has yet to be proved.

What is known is that \(|\mathbb{N}| = \aleph_0 < \aleph_1 \leq 2^{\aleph_0} = |\mathbb{R}|\) and in this paper we simply depend on the fact that there are sets with size strictly greater than aleph-zero, and we will refer to these as simply being of size aleph-one or bigger.

2.4 Extending to all Unicode texts
The first way to alter our enumeration is to extend the character set from Borges’ 25 characters to the full set of Unicode ‘code points’, which are almost like characters but not quite (Unicode, 1991). Indeed, we’re going to have the index range for these code points to not only cover the existing 137,000 or so code points, but the complete range of theoretically available code points, which is 1,114,112. Similarly, rather than limit the number of code points in our ‘books’ we will instead think of strings of unlimited, but finite length.

Therefore, pieces of text of any arbitrary length using any Unicode compatible character set will be covered by this enumeration. Any such string can be given a unique index number as follows, where \(n\)

\(^3\) As Borges himself notes in the short story itself.
is the total number of code points in the given string and \(c_p\) is the code point value for the \(p\)th code point in the string:

\[
\text{Unicode string unique ID (USUID) number} = \sum_{p=0}^{n} c_p \cdot 1,114,112^p
\]

Unlike the enumeration of books in Borges’ library, this enumeration of Unicode strings does continue for ever as \(n\) can be arbitrarily large, making this set of strings aleph-zero infinite in size.

We can construct an alternative enumeration for this same set by encoding these Unicode strings in UTF-8 format as a series of 8-bit binary numbers. This enumeration would be formulated as follows:

\[
\text{UTF8 (file) unique ID number (UTF8UID)} = \sum_{p=0}^{n} c_p \cdot 256^p
\]

Where now \(c_p\) is the number (0-255) represented by the \(p\)th byte in the file.

2.5 Extending to all binary encoded recordings

Clearly the last enumeration could be used for any binary file, but we happen to have decided that these are UTF-8 encoded files. What if the binary file encoded a video, or any other digitally recordable act of communication (such as a song, or video, or vector graphic, or motion captured gesture or whatever). To create an enumeration to cover all different kinds of encodings of binary files, we can combine together two enumerations using the fundamental theorem of arithmetic that any number is the product of a unique combination of prime numbers.

So, if we imagine an enumeration over all known encodings, then we could assign any particular encoding a unique ‘encodingUID’ number (for example \(0 = \text{UTF-8}, 1 = \text{UTF-16}, 2 = \text{MPG4}, \text{and so on}\)). Then, using a similar enumeration to that used for the UTF8UID numbers above, we can imagine giving every possible finite length binary file a unique ‘binaryFileUID’. We can then combine these two enumerations to give a new enumeration over all encoded binary files:

\[
\text{Encoded binary (file) unique ID (EBUID)} = 2^{\text{encodingUID}} \cdot 3^{\text{binaryFileUID}}
\]

The purpose of using this enumeration, rather than just the binary file enumeration is so that each index number ‘knows’ which encoding is being used by the binary file, therefore this enumeration keeps sufficient semantics about the recorded information that it could be decoded appropriately.

Note that we can imagine this enumeration being extended to cover any arbitrary (but finite) fidelity of recording. So, for example, every conceivable 4K high definition, 100 frames per second 3D, surround sound video is included within this enumeration as long as the video is of finite time duration. And the enumeration also contains all finite higher and lower fidelity copies of every conceivable finite time duration video.

So, we can now imagine a new audio visual (and more) ‘library’ that doesn’t just contain books but contains multiple higher and lower fidelity copies of every conceivable way to record information. For our purposes here, we’ll call this the “exhaustive digital library”.

2.6 Why is finite fidelity enough?

A key detail in the enumeration of this exhaustive digital library is that it contains all finite fidelity copies of any recordable information. Without this limitation it wouldn’t be possible to construct the enumeration. So, how do we know that these finite fidelity recordings are sufficient to capture any human attempt to communicate?

Well, any such attempt to communicate must be perceivable by another human and we know that the human ability to perceive and discriminate stimuli has finite limits. Therefore, for any piece of recordable information there is a finite level of fidelity of digital recording at which no human could notice a loss in information conveyed by the recording.

This is a fundamentally important observation for this paper as it is this that allows us to know that there will be no recordable human communication that cannot be sufficiently represented by some entry in our exhaustive digital library. So our digital library that is exhaustive by construction has also been shown to comprehensively cover everything we wish to hold in such a library.
2.7 An enumeration over all recordable human communications

Now that we have an enumeration that gives anything recordable a unique EBUID, we can combine this with an imagined enumeration over all humans that will ever live⁴ (giving each one a unique ‘humanUID’) and another imagined enumeration over all milliseconds since the big bang (‘millisecondTimeCounter’). Putting these together we can formulate an imagined index of Recordable Human Communication Unique IDs (RHCUIDs) for all conceivable acts of recordable communication by any human⁵ ever:

\[
\text{Recordable Human Communication UID} = 2^{humanUID} \cdot 3^{millisecondTimeCounter} \cdot 5^{EBUID}
\]

The above description of the RHCUIDs gives a basic proof by construction that such an enumeration would in theory be possible by a god-like observer of the universe.

Furthermore, the existence of such an enumeration over all conceivable recordable human communication demonstrates that the total cannon of all actual human communication that will ever happen is also no larger than aleph-zero.

Indeed, assuming that there will be a largest humanUID (even if this enumeration includes the evolutionary descendants of humans) and a highest millisecondTimeCounter value in which humans exist then there will only be a finite number of actual attempts by humans to communicate. Also, during all of that time that humans exist there will be an EBUID with the largest, finite size that would be needed in order to have recorded each communication act by humans in sufficiently high fidelity so as not to have any noticeable loss in any of the information being communicated.

In other words, assuming there is a “last human”, then to faithfully record all human attempts to communicate ever will ‘only’ require a finite description length! (the ‘only’ is in quotes because this finite number will obviously be extremely large).

3. Our recordable discourse is trapped in aleph-zero

So, what is the purpose of creating all of these rather absurd enumerations? As with Borges’ library these enumerations are of no practical use. However, even if we stay neutral on the question of whether or not there will be a “last human”, we can now confidently make the following three statements:

1. The total number of recordable communication acts that are actually made by all humans ever will be in aleph-zero (where “in aleph-zero” means less than or equal to aleph-zero).
2. The total description length of a set of high fidelity recordings of all these actual communication acts will also be in aleph-zero.
3. Even the description length of all conceivable high fidelity recordable content that humans could ever produce is in aleph-zero.

Hence, all conceivable human discourse, everything expressible by humans is ‘trapped’ in aleph-zero.

Or in other words, the fidelity with which humans can refer is only countably large.

But why is this fidelity limit worth noting?

3.1 Can’t we refer to some real numbers like pi?

The possibly surprising thing about our discourse being trapped in aleph-zero is that we have been able to discover and write about larger infinities, like aleph-one, and we are able to refer to and use real numbers, such as \( \pi \) and \( e \) and \( \sqrt{2} \) that seem to belong to sets of size aleph-one or bigger.

Similarly, the diagonal arguments such as Cantor’s, Gödel’s ( incompleteness theorem) and Turing’s (undecidability of the halting problem) seem to depend on our ability to see truths that go beyond the enumerations being used in the proofs. And, a lot of mathematics (such as calculus) depends on the fact that real numbers form a genuine continuum (in aleph-one or bigger), unlike the rational numbers (which are in aleph-zero).

---

⁴ And we’ll include in this theoretical enumeration all the evolved descendants of humans too.

⁵ We can easily imagine extending this enumeration to all agents in the universe, but humans will do for our purposes here.
So, in the realm of logic and mathematics there seems to be a genuine importance to the larger infinities such as aleph-one. This in turn suggests that the set of things that are ‘true’ in this universe is greater in size than aleph-zero. Also, given that we have discovered this mathematics of larger infinities and are able to work effectively with real numbers it is seems at least plausible that we are somehow able to think in ways that go beyond aleph-zero.

However, even if we are able to think with higher fidelity, the enumeration presented above provides a definite limit to the fidelity with which we can express ideas. And this includes every possible way that we could recordably communicate our ideas about aleph-one and beyond. So, what is going on?

3.2 What if the physical universe is ‘larger’ than aleph-zero?
Our models of physics, including quantum mechanics, use mathematics that depends on the continuous nature of real and imaginary numbers. This suggests there is also a certain physical importance to the infinities beyond aleph-zero.

However, a key unknown about the nature of the physical universe is whether or not the Heisenberg uncertainty principle describes a limit on the fidelity with which we could ever observe the universe, or a limit on the ‘fidelity’ of the universe itself. Or to put it another way, is space time a genuine continuum (in aleph-one or beyond) or is it actually countable (in aleph-zero)?

There is some disagreement among physicists and philosophers about this issue and indeed given the Heisenberg limit on what can be measured it is unlikely that this question could ever be solved empirically! But there is no need to take a firm view in this paper, as here it suffices that we note the two possibilities. Either the scale of the universe is in aleph-zero, or it is larger than aleph-zero.

3.3 An aleph-zero scale universe
If the scale of the universe is within aleph-zero, then there would be no discrepancy between the scale of what humans and computers can express and the scale of the universe itself. There would still be problems of tractability (both in terms of the time and resources required to calculate or express certain things), but there would be no fundamental fidelity gap between reality\(^6\) and expressibility.

3.4 An aleph-one or larger scale universe
On the other hand, it is possible that the scale of the universe is in aleph-one or larger. Even if we could never observe the full fidelity of the universe, it might be that space-time is indeed a continuum just like the real and imaginary numbers that we theoretically use in our models of it.

In this case there would be a fidelity gap between the reality of the universe and the expressibility of humans and computers. It would rule out the possibility of using digital computation to accurately simulate the universe at the fidelity of the universe itself. It might also give some embodiment in the universe of the kind of expressibility ‘gap’ exposed by logical diagonal arguments, such as Gödel’s incompleteness theorem. Any mode of human or computer expression could only reach countably many ‘truths’ and yet there would be circumstances where what we are trying to express refers to a feature of the universe with aleph-one or more ‘truths’.

Indeed, if humans couldn’t possibly express all of the truths of the universe, then this would give an additional reason for the postmodern concerns about the gap between what we can express, what we mean to convey and what is understood by others. Or to put it another way, the idea that all human recordable discourse can be enumerated should not be seen as giving support for a modernist, totalising form of rationalism. Quite the reverse. It is a limit within which we are ‘trapped’ and the full truths of the universe might be beyond our expressible grasp.

And yet there is still the apparent conundrum of how we are able to use the concepts of real numbers and the aleph-one infinity (and higher) if all our expressions are trapped in aleph-zero.

3.5 How we refer to pi (\(\pi\))
Pi is probably the irrational real number that has been studied in the most detail. In 2016 the record for enumerating the digits of pi stood at over 22 trillion digits (Trüb, 2016). Ironically, even listing out

\(^6\) In this paper we’re admittedly working with a very simple, naïve realism.
all these trillions of digits would be a slightly less accurate way to refer to pi than writing its well known symbolic signifier: `π`.

With this one character we refer precisely to an infinite series of digits because we know the algorithm with which to calculate any arbitrary precision of π. It is a non-halting algorithm, so the expansion from the symbol, π, via the algorithm, to the infinite series of digits is intractable, but that does not render our ability to refer inaccurate.

Furthermore, in mathematical formulae we can use the symbol π to stand in for the exact number and in some situations different uses of π will cancel each other out.

So, we can refer to π accurately because although it is irrational we can describe it accurately with a finite length algorithmic description. We can then meaningfully refer to this description (and thereby the accurate number) via its even simper, naming signifier.

Note, however, that the description, “the next real number bigger than pi” does not refer correctly to anything as there is no ‘next’ operator on the real numbers. So, not all attempts to refer to a real number using a description will be successful. However, the conjecture here is that for all real numbers (rational or irrational) that we can refer to accurately we will be doing so via a finite length description or algorithm of some kind.

As each of these descriptions will have at least one EBUID that encodes the description, therefore there are only countably many irrational real numbers that we can accurately refer to in a similar way to which we refer to π. Obviously the other well known irrational real numbers, like e and √2, are members of this set, but the conjecture here is that all irrational numbers that we can refer to accurately must have a finite description length algorithm that can generate any arbitrary precision expansion of that irrational number (given enough computational time and resources). Furthermore, the size of that set will be aleph-zero, it will itself be a countable set.

Conversely, the overwhelming majority of irrational numbers cannot be referred to accurately with a finite description length. These numbers require all of their infinite series of digits to be listed in order to be accurately referred to and to write down such an infinite signifier would be an intractable undertaking.

In other words, we can only actually work with a countable set of irrational numbers. Our descriptions of these numbers and of aleph-one and so on are all finite length descriptions, and this is how we are able to usefully refer to aleph-one and some irrational numbers within our countable set of all possible acts of communication.

3.6 How we recordably refer to unrecordable thoughts, feelings and experiences

We often use language to refer to experiences that many people will have shared, but for which the full content of the experience could never be recorded or communicated in its entirety. Just as with irrational numbers, like π, we use descriptions to indirectly refer to the otherwise inexpressible. For example, we may use a description that tries to convey to the reader which of their own unrecordable experiences we are trying to refer to. Or, if the reader has never had such an experience we may try to describe how the reader could get into the right kind of situation in which they would experience something similar. We then use short signifiers, like ‘pain’ or ‘love’, to refer to the experience ‘via’ a lifetime’s collection of these longer descriptions of what, say, ‘love’ is. However, we could never communicate the experience itself. And our descriptions of experiences are never (or rarely) as accurate and repeatable as our algorithmic descriptions of specific irrational numbers like π.

The point being made in this paper is that the total conceivable cannon of all such recordable descriptions of unrecordable thoughts, feelings or experiences would also have to be ‘in’ aleph-zero.

3.7 Comparing humans and computers

The purpose of this paper is not to make the claim that humans and computers can refer to the same things in the same way. When a human who has bitten into an apple refers to this experience via the description, “it was just like biting into an apple” they are referring in a way that no non-human could ever fully understand or achieve. A robot that can bite apples might have an experience when doing so, but this would not be a qualitatively similar experience to those had by humans. Therefore, if the
robot were to say, “it was just like biting into an apple” it is not quite referring to the same thing as a human would be. If an unembodied, text-analysing AI (that doesn’t have the capacity to even potentially experience biting into an apple) were to ‘say’, “it was just like biting into an apple” then it’s questionable whether or not the generated text describing an experience is meaningfully referring to anything.

But this paper is not examining the general question of how recordable communication (like text) can meaningfully refer. Rather the purpose of this paper is to highlight that the difference between how computers could potentially refer compared with humans is not a difference in the scale or fidelity of communication as both are trapped in aleph-zero.

And so, if we’re looking to understand the difference between how humans and computers can meaningfully refer we should not think in terms of humans being able to refer to ‘more’ than computers, or that humans could refer in a ‘finer grained’ way than computers. Any difference must come from elsewhere, such as the difference between the ways that humans and computers are engaged with the world around them.

Note that to motivate these conclusions we do not need to claim that human thought is necessarily bounded in a similar way to human expression. It may turn out that human cognition taps into quantum computation in a meaningful way and so is vastly superior to traditional digital computation. However, even if this were true in an interesting way, our ability to communicate ideas to each other would still be bounded by countability as laid out in this paper.

It is similarly worth noting that even if analogue computing or quantum computing are able to work in aleph-one fidelity by exploiting aleph-one features of the universe (if they exist) any attempt to communicate the ‘results’ to humans would again be bound by the aleph-zero countability constraints discussed above.

4 Use of Unique IDs (UIDs) in programming

Finally, we’ll take a quick look at what the preceding discussions imply for the appropriate use of unique IDs (UIDs) in programming environments such as in Semprola (Sharpe, 2018).

For any attempt to improve the semantic depth of programming to get closer towards the level of meaning imbued in text and other forms of communication by humans it would seem, at first glance, that the programming environment should be ever more like something that humans would normally work with or be ever more biologically inspired. And, humans do not normally use UIDs in their daily lives nor is there any hint of any suggestion that anything like UIDs are in operation within the mechanisms of the brain.

So, the use of UIDs feels at odds with any project to improve the semantics within programming.

However, as has just been discussed above, the entirety of human discourse is enumerable and therefore it would be possible (in theory) to assign to every recordable act of communication a UID along the lines of RHCUID above.

For most of history we have had no mechanism or indeed purpose to do anything like this, but with the emergence in the last decades of digital communication technologies and indeed a growing number of people “life logging” all recordable aspects of their own life, it is the case that a growing number of communication acts by humans are actually being given UIDs even if they do not belong to a single, universal enumeration.

All phone calls, text messages, Skype calls, emails, documents, photos and more each have some form of explicit or implicit UID. Indeed these UIDs are invaluable to ensure that the identity and thereby history of a recorded act of communication remains stable even as the binary file encoding that communication act is copied and transported around a multitude of computer infrastructure.

As mentioned in relation to Borges’ library in section 2.1 above, it is the history of a piece of text (or encoded binary file more generally) that is the vital link between the true semantics of the text with its original author. Without this link a randomly generated piece of text only has the appearance of signifying an intended meaning.
With a physical book the physical object itself can act as the maintainer of the continual identity through which the history of the book to an author can be traced (or more usually assumed). But with virtual bits of text this ‘metadata’ relationship has to be explicitly maintained with the text if it is not to be lost. This is one of the crucial roles that UIDs can perform for virtual text or, of course, any binary file recording of an actual communication act. This is one of the reasons why Semprola ensures that every piece of text (for example) has a Semiotic Programming Unique ID, SPUID.

There are of course other, important ways that UIDs can help maintain high levels of semantic information (such as by helping distinguish the identities of relata with greater semantic accuracy than would be possible by just using human readable text labels), but these are not so relevant to the discussion here.

Having explained the reasons why a programming environment like Semprola would systematically use UIDs, it is still possible for this abundance of UIDs to give an impression that what Semprola will be capable of is so obviously countable in scale that this must somehow be less than what humans are capable of. Hence, the key message of this paper was that actually all of human communication is also only of countable scale, so this apparent ‘limit’ on an environment like Semprola is no greater limit than already applies to humans!

5. Conclusion
In this paper we have compared the fidelity of the universe with the fidelity with which humans and computers can communicate recordable ideas. With the use of various enumerations it was shown that humans, just like computers are only able to communicate to a fidelity equal to or less than the aleph-zero infinity and so there is no fidelity gap between humans and computers. And yet, it is possible that the universe has an even greater fidelity of aleph-one or higher. If true this would imply that there is a gap between what could ever be communicated by humans or computers and the scale of things that are true about the universe.

However, a key point of the paper is to highlight that there is no such ‘scale’ gap between the fidelity with which humans and computers can refer. So, if we wish to understand the differences between how humans and computers refer we cannot simply suggest that humans can refer to ‘more’ than computers or that humans can refer “in more subtle ways” than computers. Instead, any gap between the ways that computers and humans can meaningfully refer must lie somewhere other than scale.

6. Appendix
A lovely online version of the Library of Babel has been constructed by Jonathan Basile (https://libraryofbabel.info). Not only can you virtually browse through the shelves in the hexagonal library, but due to its computational construction, you can also search for books that contain particular pieces of text. Below is a list of four such books where the abstract of this paper can be found on a page in their random text:

- Page 115 of book at location: https://libraryofbabel.info/bookmark.cgi?ppig2018.1
- Page 15 of book at location: https://libraryofbabel.info/bookmark.cgi?ppig2018.2
- Page 179 of book at location: https://libraryofbabel.info/bookmark.cgi?ppig2018.4

7. References
Cantor, G (1891) via Wikipedia https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument
Trüb, P (2016) 22.4 trillion digits of pi. https://pi2e.ch/blog/
What Lies in the Path of the Revolution

Antranig Basman
Raising the Floor - International
amb26@ponder.org.uk

Abstract
Increasingly, the rights and capabilities to own technological artefacts, where they exist at all, are reserved to corporations and not to citizens. There are historical, economic, metaphysical, ideological and cognitive reasons for this situation, in addition to purely technological factors, which we will trace by following the fate of various concrete examples, analysed into five categories of ownable elements. These categories are those of ownable function, ownable expression, ownable data, ownable installations and ownable economies. In this paper, we attempt to align these goals of ownability into a research and activism program by describing a set of revolutionary goals in each category, and tracing ways we could reach them.

1. Introduction
One of the principal effects and aims of the 1789 French Revolution was the dismantling of France’s feudal system, under which some citizens (nobles, officers of the church, etc.) enjoyed special rights, protections and income, while others (serfs, peasants) faced special taxes and in some cases were prohibited from holding personal property at all. We argue that a form of “digital serfdom” has rapidly grown up around us, where various important classes of artefacts, increasingly essential for everyday life, including participation in political and economic life, cannot be effectively owned by ordinary individuals. The power to own these artefacts is increasingly reserved only to corporations, which as a result of legal frameworks arising from the end of the 19th century, especially in the Anglo-Saxon nations, are considered legally as persons.

However, the barriers to ownership are not merely legal, but cognitive, economic, technological and even metaphysical.

Many of the technological barriers to ownership, especially as regards software, can be seen as embedded in certain questions of “reuse” — one of the central affordances of ownership is the ability to transplant a thing from its original location to a different one, following the desires or person of the owner. For most kinds of software this is possible in only a crude way — an “application” can be installed on one machine rather than another — and with the rising prevalence of rental or cloud-based models for the deployment of software, it is decreasingly possible at all.

In this paper, we use “ownership” to refer to a set of related use patterns where owners can pursue alternatives, repurpose and adapt their property, or maintain their property as the surrounding world changes. We will survey the kinds of things that we might want to own and how we could set about transforming our environment so that they could be owned.

In the rest of this paper, we step through examples of different technological artefacts occurring in the ecosystem encompassing operation, installation, and development of software: we describe actual interactive interfaces and tools as artefacts of function, the underlying source code as artefacts of expression, the content manipulated through software as data, the machines and networks running software as installations, and the means for organising labour around software as economies. For each of these, we describe examples of hypothetical ownable use, and then analyse why enacting those examples is today impractical or impossible.

In terms of aligning with a historical trajectory, we should note that even if we envision a socialistic far
future, establishing the means for personal property and thus ending feudalism is a basic requirement to
even operate capitalism. Marx did not even propose to operate communism without a basis for personal
property.

It would be supremely easy to make our point about poor ownership values by picking on the output of
one of today’s major corporations. For example, we could refer to recent versions of Adobe’s “Creative
Suite” which is now no longer ownable even nominally, since these are exclusively available through
leasing models. Or we could refer to Microsoft’s or Google’s apparently “free” suite of office tools
which are readily available in the cloud, but are only ownable on the basis of offering our personal data
or our eyeballs to advertisements in exchange for them — these services may be unilaterally withdrawn
without recourse at any moment, and are the touchstone of today’s digital serfdom.

We will take such problems as read — we are not seeking to restrict choices of operating models by
corporations, but instead to increase choices that are even in theory available to citizens.

2. Ownable Function

If you find a piece of software that does something of value to you, it should be possible to make it your
own. This implies that you can keep using it as part of the collection of tools you carry with you, in
familiar contexts, or experiment with using it in novel contexts. In this respect it would be like a tool
that any workman or craftsman applies to their work. We’ll illustrate the kind of thing we mean, with
respect to the software of today, firstly with a small-scale example, and then a larger-scale example.

2.1. A small-scale example: colour picker

Figure 1 shows a colour picking tool designed by a particular community. It is a competent if unexciting
example of its breed. Imagine that you had got particularly used to the layout, idiom and affordances of
this colour picking tool and wanted it to be your colour picker of choice in some other context, such as
formatting a text document, and wanted to know how to achieve this. People unfamiliar with the nature
of today’s software ecologies would be deeply surprised to learn that the answer to this question is that
it is economically impossible. For comparison in Figure 2 is the standard Windows colour picker, as
being invoked from a popular code editor, Eclipse, which is written in the Java language. It is visually
clear that they perform the same basic function, and that they are even capable of sharing the same

---

1“We by no means intend to abolish this personal appropriation of the products of labour, an appropriation that is made for
the maintenance and reproduction of human life, and that leaves no surplus wherewith to command the labour of others.” (Marx
and Engels, 1848)

2As numerous of Google’s products have been in the past: https://en.wikipedia.org/wiki/Category:
Discontinued_Google_services

3An open source geographical information system named QGIS
representations for the colour value\textsuperscript{4}. What could be the source of the difficulty in dropping in one in the place of the other?

A naive user expecting the colour picker to be an ownable artefact might discover that it cannot pick or drop colours outside the window of the host application. We could imagine them closing the host application expecting the colour picker widget to stick around, instead of disappearing. In practice, these kinds of problems require users to carefully mediate between applications, e.g., using the host application of the colour picker as a transit station where colours to be sampled are imported by screenshotting, and exported by copy-pasting e.g. the HTML notation for the colour.

These barriers to ownable use originate in development choices outside even the expert owner’s control. Our example colour picker originates in an application, whereas a luckier user might have access to and prefer a colour picker hosted at the operating system level, which may travel between different applications. On a deeper level, these problems are embedded in the stack of technologies used to implement and run the colour picker, which we return to in section 3.1.

The economic reuse value of this widget across this boundary is zero, a poor compensation for the hundreds of hours of expensive developer time that went into producing it. As we outlined in our introduction, there is a close relationship between the faculty of ownable function, and the faculty called “reuse”, which software developers talk about a lot. In fact, one faculty is a subset of the other — while “reuse” is a phenomenon which may be encountered when swapping out elements of a system which are in direct contact, ownability also brings into consideration elements of the system that form more distant context.

2.2. A large-scale example — iNaturalist

\textit{Figure 3 – A view of one contributor’s observations on iNaturalist}

\textsuperscript{4}The range of realistically useful alternative colour tools is in fact much larger. Jalal et al. (2015) study how expert designers invent new mechanisms and contexts to manipulate colour, and argue that digital colour manipulation has been wrongly reduced to a simple problem of selecting and storing colours in contextless palettes.
Our chosen large-scale example is iNaturalist\(^5\), a platform which is free for use by any citizens wanting to contribute their observations of living things, which are categorised by species, confirmed by expert enthusiasts, and plotted on a geographical database. This is an open source application written mostly in the Ruby language, with the significant admixture of JavaScript essential to any modern webapp. Figure 3 shows a typical view of this web application in use — one expert contributor is viewing the first page of their own contributed observations to the system. We will adopt this application as a “running example” through our succeeding sections on ownable expressions, data and installations.

iNaturalist is an application developed to serve the public good, whose source code is readily available, well-maintained and has a welcoming community. This should offer us a radically different perspective on ownability compared to commercial closed-source software such as the Adobe Creative Suite, but as we will discover, attempts at ownable use have very much the same results for the citizen: ownability doesn’t come at a price they can afford.

In our example, an expert user of iNaturalist wishes to adapt this existing, rich platform to assist their own projects by making a number of additions to its functionality and interface. This example is sourced from a real expert contributor. The expert has a general familiarity with the popular statistical analysis packages which are used in their own field, and an everyday acquaintance with web technologies, but is certainly far from the skill-set of a professional developer. We will survey in the following sections how the economics currently play out for the assistance that our field can currently offer them, and then lay plans for how a revolutionary uprooting of its metaphysics could lead to a better story. We’ll also survey a few early “green shoots” of existing starts towards better authorial values and consider how they could be promoted and supplemented with other approaches to eventually form an integrated forest of ownable technology.

2.2.1. Planned Adaptations

Our expert wants to adapt the iNaturalist platform in (at least) the following ways:

- To be able to contribute lists of species that are considered to be available in particular locations, and to associate those lists with geographical boundaries drawn in a free-form way on maps
- In addition to these lists being curated in an on-going way by assigned “project managers”, each list will contain fields in addition to those maintained by the iNaturalist platform — e.g. the natural “habit” of a species, whether it is bush-like, tree-like, etc.
- While the core iNaturalist interface allowing the contribution of new observations and viewing existing ones should remain in place, certain new views should be available — for example, to select two or more lists and visualise their intersection or difference in some compelling graphical terms, e.g. a sunburst diagram

while many more examples of interesting adaptations could be listed, the above are sufficient to stress all the issues of ownability we are interested in exploring here.

The core ownability issues relate to how shared function is to be shared, how it is to be developed and tested, and at what cost, and how it is to be deployed. The expert’s need for “ownable function” in this case goes beyond that of a craftsman merely contemplating a single “ownable hammer” to put in their workshop. In this case we are imagining function shared between overlapping communities, in the structure that Suchman (2002) describes as “multiple, located, partial perspectives”. Our expert wants to plan for related or unrelated communities structured around other experts, each with their own perspectives on what species (there) are, and what aspects of them are relevant to their communities. These all need to coexist with the “base function” offered by the core iNaturalist platform itself, and its base community. How this is done in practice has implications for the expressions of the authors assisting the expert, the expressions of the core community, the data they work on, and how the resulting networks of cooperating, actually installed software are constructed, which we will consider in succeeding sections.

\(^5\)https://www.inaturalist.org
3. Ownable Expression

It should be possible to own the means to create and modify software, which we currently call programs. At a small scale software is built up out of collections of authorial expressions. These expressions describe the kinds of things that can exist, command particular selections of these things to actually exist or not exist, or change their properties while they do exist.

For these expressions to be ownable, all authors should have the ability to freely contribute their expressions to the work of others, and freely “buy into” and “buy out of” the expressions of others. We have formulated this requirement as the Open Authorial Principle: “Any expression by one author can have its effect replaced by an additional expression by a further author” (Basman et al., 2018). When we say that this “should be possible”, we imply that it should be possible in practice — that is, that the authors’ expressions can be successful at a cost which is affordable. This is a statement as much about economics as it is about engineering. These costs can be measured in numerous terms, for example, investment in a particular skill-set, direct monetary terms, effort in building a relationship with a particular community, or cognitive effort.

3.1. Small-scale example: colour picker

We return to our colour picker example from section 2.1 and the alternative in Figure 1, which a user wishes to reuse instead of the standard choice (Figure 2) that is brought up by an interaction in their application.

Given that our user is or has recruited the assistance of a developer, the first road block to performing this adaptation is to penetrate to the inside of a delivered application to locate the colour picker to-be-substituted. Second, the developer must reconcile the form of the migrating colour picker with the expectations of its new environment. Added to the difficulties created by the mismatch in implementation languages (C++ on the left, and a sandwich of Java and C on the right), these come together with different idioms for interpreting the contents of memory, and even for addressing and painting areas on the screen.

We note here that while some heroic efforts might succeed in transplanting this harmless-looking colour widget, the resulting assembly would be little more than a curiosity — the hapless user would effectively have taken on the responsibility for maintaining their own personally customised version of the entire application.

To add insult to injury, the techniques for implementing and delivering software create impenetrable boundaries from the point of view of the user as well as the developer, as we described in section 2.1.

Sometimes designers explicitly plan for this kind of reuse, for example by creating “plugin” architectures where a particular well-characterised part of a system is designed to be replaceable by alternatives. The converse is a “component” architecture where the one part of a system is free to appear in many different locations. Biddle and Tempero (1998) name these two kinds of reuse as “context reuse” and “component reuse” respectively. Where the designer has not laid plans of these kinds, developers, integrators, and users are out of luck — especially where language or platform barriers make reuse impossible. Even in the case where these kinds of reuse are planned for, they usually result in a strongly bounded ownability where creators and users of plugins commit economically to one particular environment over others.

The fact that the user has to be troubled by these irrelevant considerations of platform betray that we are in a pre-industrial environment plagued by irregularities like screws that are not of a size to match screw threads, trains that cannot run on each other’s tracks, or electricity that may not travel between different wire networks (David and Bunn, 1988).

The ultimate implication of these issues is that programming languages as they are currently constituted need to be abolished, through a process of being dissolved into an “integration domain”, a construct described by Kell (2009). An integration domain is a space in which one aligns multiple artefacts to co-operate, and which, importantly, is the only space that may contain assumptions about the particulars.
of several artefacts. This implies that artefacts to be integrated expose structured representations of themselves (Clark and Basman, 2017).

In this example, the integration domain should allow a developer to substitute the existing expressions that connect the colour picker in Figure 2 to its operating context for a set of similar expressions mapping to the picker in Figure 1. Tasks for an even more ambitious integration domain would be to bridge these kinds of divides across multiple, separated but cooperating devices rather than just across process, language and integration boundaries within a single machine.

3.2. Large-scale example: iNaturalist

![Figure 4 – The source code for iNaturalist in GitHub](image)

The hypothetical process of adapting iNaturalist follows a similar procedure to the previous example: determining the sites where changes are to be made, then reconciling existing expressions with the new features we want to add.

3.2.1. Correspondence between expression and function

In Figure 4, we show the root page of iNaturalist’s source code, the primary representation of its authorial expressions, as shown in GitHub, a very widely used tool for developers to share and manage their
code. The first thing to note is the total lack of correspondence between this view and the view of the application in use shown in Figure 3. Again, those unfamiliar with the choices made by those who designed our programming idioms might be deeply surprised to learn that this isn’t just a superficial phenomenon — in fact, there is no reliable way, even with the most powerful analysis tools available, to determine for example given the view of the “Intermediate Hook-Moss” observation in Figure 3, which part of the source code in one of the numerous deeply nested directories shown in Figure 4 is responsible for it. This is nothing to do with the particular technology choices made by the developers of the application — it is instead a fault in our entire paradigm of software construction. The particular choice, of implementing a Ruby on Rails app with a fairly rich JavaScript front end, represents a mainstream, conservative choice of a well-supported and generally well-liked technology stack, but any other credible technology choice would have led to just the same result.

On the other hand, those who are indeed experienced in software construction would find this point so obvious as to take it completely for granted. Anyone inducted into the Eleusinian mysteries of code takes on board as implicit knowledge that they will have to acquire a broad spectrum of “jungle skills” which help them navigate these kinds of informational swamps — not least, an in-depth study of the structuring conventions used by the particular technology in question, but more broadly applicable techniques such as deliberately banging on things to see what breaks, studying the structure of long stack traces, or searching across the codebase for the occurrence of tell-tale strings.

In (Basman, 2017) (section 6.3) we present an alternative idiom for building software which would substantially bridge this kind of problem. Rather than building software with opaque “machines” such as function calls and objects, we would instead use reversible elements named “lenses” to express and enforce relationships between their parts. These would act both “vertically”, enabling any part of the user interface of the application to be traced back to its source expressions, and “horizontally” in order to structure data flow around the application itself. There exists a mature community (Bx, 2018) devoted to the study of such reversible elements under the name of “bidirectional programming” but its activities are confined to establishing proofs of esoteric theoretical properties of such elements rather than constructing practical alternatives to general-purpose application development.

3.2.2. Sharing expressions between communities

In order to effectively share the expressions leading to the kinds of adaptations that our expert desires in section 2.2.1, we incur difficulties in other areas of ownership as well. While we will specifically treat problems of shared installation in section 5, some of these will be incurred during the process of developing shared expression and design and so are addressed here.

Firstly, we would like to consider how the expressions (source code) implementing the adaptations will coexist with the ones shown in figure 4. This involves social, design as well as technical and economic issues. To start with, these are adaptations which may not be appreciated by all users of the platform. A typical design solution to problems like this is the kind of “giant ribbon” familiar to all from modern office applications, where every conceivable function is packed into a crowded menu or ribbon widget, each represented by a very tiny icon in the hope that the presence of irrelevant functions is as small a distraction as possible. Some of the forces behind such designs are the difficulties in arranging for the expressions forming the adaptations to act in a suitably context-aware way, presenting themselves to some communities and not others.

Setting aside this design issue, we have a meta-design issue, in that we couldn’t expect the adaptation to instantly appear fully-blown in its final form. More realistically, we expect an extended process of co-design, where competing alternatives can be evaluated together with the communities of interest. While simple prototypes and mockups might be suitable for early stages of this process, it is inevitable that the later stages of co-design will have to centre around patterns of real use of a functioning system with real data. The question is — what system, based on what expressions, operating on what data, installed and maintained by whom?

The standard answers offered to these kinds of questions in communities which can currently be sup-
ported rely on some kind of “centrism”. The community wanting to make some adaptations must start to build a relationship with the core community. Modern distributed source control systems such as git reduce the formality of the early stages of this considerably. Since this is an open source project, the adapting community can simply “fork” the source code shown in Figure 4 and start modifying it. Beyond this point, however, the costs start to escalate rapidly, along two dimensions:

- Unless the adapting community wants to commit indefinitely to maintaining their own adapted version of the artefact, they will need to build a strong relationship with the upstream community, and become familiar with their coding guidelines, administrative practices, and ensure that their adaptation is structured in terms that they are prepared to accept and maintain
- During the process of design, they will need to become familiar with the deployment and integration requirements for running their own installation, as well as considering what data sharing idiom is appropriate between their installation and upstream.

These profound problems of design and expression, which should seemingly be our most urgent ones, are not considered interesting problems by academia, and are not considered profitable to tackle by corporations. They relate to what could be called a “commons of the mind” and fall to whatever resources society as a whole has to devote to such issues, which are minimal. The means by which these problems are solved in practice can be compared to Suchman’s “articulation work”, the often invisible work to actually make technologies work together inside a community of practice full of existing artefacts (2002).

Where they relate to problems of expression itself, we trace these problems back through the intellectual history and culture of the computer science and programming communities in (Basman, 2017) and (Clark and Basman, 2017).

4. Ownable Data

As with the situation with “ownable expression”, there is both a formal and an ultimate aspect to the ownership of data. Firstly there is the possibility of physically laying ones hands on the bits involved, at an acceptable price, or for free. This formal sense of “open data” is widely acknowledged (although not universally practiced), and has a parallel role to that of “open source” in the category of ownable expression — it is the most basic “entry ticket” to ownability but doesn’t necessarily get one very far into the venue.

The more ultimate aspect of ownable data is whether the data can in practice be put to the effective purpose its owner requires. This may include

- Whether it can be transmitted to or from a place of use in a format that is intelligible at both sites
- Whether it can be reconciled with data from other authorities which may use different standards for provenance, different ontologies, different schedules and policies for validation, or have a mismatched scope
- Whether legal or intellectual property issues effectively prohibit the data being put to the owner’s purpose

Again, as with ownable expression, this is an economic proposition, rather than a formal or logical one. These capabilities must be delivered at a cost that the owner can afford.

As a corporate example, Google is slightly ahead of some rivals in providing formally open data. Google in theory provides the facility to export your data from its services using “Google Takeout”, a product of the “Google Data Liberation Front”. Facebook also allows you to download some proportion of the data it holds on you. However, given this data, it is not always clear how its value may be liberated:

- Some services such as YouTube and Facebook have ended up as natural monopolies, and so there is no other equivalent platform on which one’s data may be used
- The damage may already have been done to its capacity for use by having been incorporated under the service provider’s licencing conditions.
Balkan (2017), further, notes that in Facebook’s case, the data in question is about the citizens we are planning to empower with ownership. In this case, it is yet more crucial to empower these citizens to express and enforce the boundaries of ownership where the material constitutes the extension of themselves into a digital realm. He calls for an infrastructure to achieve this which is funded from and owned by the commons.

4.1. Ownable Data Curation — The Problem from the Front End

Let’s return to our running example of adapting iNaturalist, which has many interesting connections with ownable data.

Firstly, at the level of the expert himself, the adaptation we plan in section 2.2.1 requires the entry and maintenance of the species lists. In practice, this is something that the expert already does for themself in a tool of their own convenience. In this case, the tool is the Macintosh-specific spreadsheet named “Numbers”. The standard workflow that is available for this kind of situation is to ask the expert to manually operate some kind of CSV export functionality within the application, to upload this file to the installation server, to there operate some format conversion as part of a build or ingest process, etc.

This clunky workflow is obviously hugely unsatisfactory. The adaptations to iNaturalist should include an interface for the expert (and other users sharing the adaptations) to enter and manage data such as the species lists live. This presents quite a few problems, and a range of choices none of which offer favorable economics in today’s ownability landscape.

One approach might be to embed some suitable data management UI and persistence mechanism within our designed adaptation, and have the expert use that. One might think that freeform “spreadsheet-like” interfaces for the management of arbitrary tabular data could now be picked off the shelf in a variety of competent open source incarnations, with adaptable styling, interaction idiom and layout, ready for embedding in any online interface of choice — given the basic interaction idiom was already mostly perfected in the 80s. However, the fundamental difficulties in developing ownable function and expression that we described in the previous sections have prevented the democratisation of this capability as well.

In addition, there are further difficulties raised by available algorithms for collaboratively working on shared data, for example under the heading of operational transforms (OT⁶), “Conflict-Free Replicated Data Types” (CRDT), etc. These are hard to implement in an ownable way as a result of their great complexity, and proceed from the faulty assumption that there is always a magic “right answer” to every question of conflicting updates that never involves any further user interaction and might involve losing one user’s data.

Some work on a much more straightforward solution to a restricted class of this problem based on modelling practical intentions of data owners appears in (Haverbeke, 2015), although the complexity still isn’t negligible, and the algorithm is optimised for document-structured rather than tabular data.

\[\text{Figure 5 – EtherCalc’s rather bulky interface for collaboratively editing tabular data}\]

One notable open source and freely hosted collaborative spreadsheet editor is EtherCalc\(^7\), whose interface is shown in Figure 5. However, this is an extremely heavyweight product actually of substantially greater complexity than iNaturalist itself. Since it has been built with a standard software development idiom, it is the product of an unownable software tradition, it will be prohibitive to adapt it to lightweight use embedded within an interface rendered by another application written in a different set of technologies, such as iNaturalist — which, naturally, suffers exactly the same problem in return. EtherCalc is simply “a collaborative spreadsheet application running on the web” and offers no useful user-directed dismantling of the expressions which led to its creation — although of course it offers the standard dismantling of its expressions into insulated implementation modules, each performing a specialised task of meaning only to developers, as is considered the standard virtuous behaviour in a traditional community composed of developers.

Faced with this uncompromising landscape, our expert currently falls back on embedding views of their data using the flagship tool of the corporate oppressors du jour, Google Sheets. This platform also offers some support for querying data and establishing live connections via a reasonably standardised API, although of course offers no ownable installation values at all — as with every Google product, it may one day disappear or change without recourse. It also has a lack of facilities for naming, archiving and ensuring the integrity of particular versions of the data it manages, which is a problem we will look at in the next section.

Another possibility might be to use whatever expression-hosting capacity is present in the expert’s client tool, Numbers, to write some form of custom “macro” that periodically synchronises its data with some ownable server in the cloud. However, it was designed without such democratic aims in mind, and in any case would not solve the curation and versioning problems that we now turn to.

4.2. Ownable Data Curation — The Problem from the Back End

Widening our focus from our iNaturalist example for this section, let’s pass on from looking at the problem from the front-end perspective, and look at what supports there exist for building networks of cooperating curators of live, version-managed open data. One important organisation in this area is the Open Knowledge Labs\(^8\) which have an initiative known as Frictionless Data\(^9\). It tackles the problem of ingesting data from various popular housings (including the basic CSV files we recommend our expert to produce using our “clunky workflow” of section 4.1), and packaging it with sufficient metadata as a “Data Package” that may be safely forwarded into a variety of standard kinds of repository, including the git repositories we saw in section 3 as well as JavaScript’s standard NPM registry, etc. This will certainly be sufficient for the modest-sized datasets that our expert is likely to generate directly in a UI, although this idiom may fall down for extremely large or continuously updated datasets such as sensor readings. In any case, it is a valuable “next hop” for our data even if, as we explored in the previous section, we lack any particularly competent user interface that would help our expert get the data in.

4.3. Back End Ownable Data for iNaturalist

The iNaturalist community already has substantial experience with data federation. As well as having several active localised downstream communities managing separate iNaturalist installations for Canada, Colombia and other nations, it also has several upstream communities, for example the Global Biodiversity Information Facility (GBIF) with which it shares observation data, and the Catalogue of Life, with which it shares taxonomic data. However, this federation has required significant maturity in the related communities and still happens on a somewhat ad-hoc, batched basis (currently updates are sent every week to GBIF\(^) through specialised formats and protocols designed just for these communities. For example, Figure 6 shows part of the output of a JSON data feed from the Catalogue of Life, its taxonomic data held on the Lesser Black-Backed Gull \textit{Larus Fuscus}.

Towards the bottom of the figure, you can make out the beginnings of a taxonomic tree classifying the

\(^7\)https://ethercalc.net/
\(^8\)http://okfnlabs.org/
\(^9\)http://okfnlabs.org/projects/frictionless-data/
species in question. This kind of data is central in interpreting the “lists” that our expert and their peers may manage; they may subscribe to the classification in this catalogue, or another, or instead have their own opinion on how it should be organised. To support these activities there should be a choice of usable off-the-shelf means to subscribe to and submit amendments to data in such feeds, and the feeds themselves need to be organised to permit this.

What is our subject-matter expert to make of output such as that in Figure 6? Their enterprising nature tends to mean they can make a fair amount of it, but there are obviously many hurdles to be crossed in turning such “formally open” data into something of effective use. To start with, it helps simply that, rather than being notated in some old-fashioned, bulky encoding such as XML, it is made available in JSON, the natural format for the universal web language, JavaScript — from here it can be directly manipulated by popular data visualisation packages such as D3, Vega, etc., as well as the primitives of JavaScript itself. Some might contend that this is a mere surface phenomenon, but diSessa (2000) is clear that even small incremental notational improvements over the centuries have been responsible for vast cumulative effects in promoting thought and innovation.

However, it is also clear that there is a huge scope here for supporting our expert in the great mass of “articulation work” (Suchman, 2002) needed to put this data to use. For a start, it is clear that this data is part of a wider referential whole — the document contains embedded references to numerous other related documents in an essentially opaque way. Secondly, such documents typically require significant reorganisation, filtering, and tidying before the elements of interest for a particular task can be brought into sharp focus. For these tasks we imagine employing again the “lenses” that we spoke of in section 3.2.1 which reversibly image such documents into the focused elements of interest. This “dialect of lenses” would itself naturally need to form a domain of “ownable expression” of the kind that we motivate in section 3.

There are many articulation annoyances in lining up the data in such feeds with our expert’s needs, but in the next sections we’ll consider more profound issues related to how these feeds are constructed in the first place.

4.3.1. Feeds are Not Lenses

Firstly, there is the fact that these feeds themselves don’t constitute any kind of lenses in themselves. Data is pushed out through them, but it is received back in by another, generally ad hoc process. Given the Catalogue of Life is a heavily curated dataset, this is no great surprise since any updates should be vetted by expert human eyeballs before being accepted. But nonetheless, even the process of requesting an update doesn’t take the same form as the feed itself. To some extent this is a reflection of the fact that
such systems (including iNaturalist) are implemented with previous-century persistence technologies such as SQL, and so the process of assembling a feed as a “view” is naturally uninvertible. More modern systems such as the CouchDB family of databases that we refer to in the next section 5 on ownable installation have partial answers to how updates propagate between databases, but these are still framed as forming part of the implementation level rather than the user level of the data network.

4.3.2. Ownable, Public Version Management
Secondly, there is the issue of version management. Since the Catalogue of Life is a highly mature public dataset, it has a well-formalised approach to issuing validated versions of the data, by publishing validated Annual Checklists in addition to a constantly evolving Dynamic Checklist. However, this is an ad-hoc system designed for a particular community, with its details encoded into the URLs of the feeds in a corresponding ad-hoc way. It would not be possible for a user to trace the lineage of a particular piece of data to discover how and when it had been introduced, or what previous versions it appeared in, without a costly manual procedure. The process of data version management needs to be as ownable as anything else — with the possibility to “take away” entire historical lineages of data with their sequencing and provenance relationships intact.

4.4. Plans for Ownable Data
To collect together all of our threads from the above sections, there is clearly scope for a great deal of valuable work in this area, producing an ownable platform supporting the kinds of data entry, encoding, transmission and reorganisation tasks that we have toured above.

Many of the version management issues that we raise in section 4.3.2 have credible solutions in the elite tools which developers have produced for their own needs, such as the git system we refer to in section 3. As we mention in section 4.2, in combination with some packaging conventions derived from the “frictionless data” community, and the use of further conventions on structuring mutually referential islands of data encoded in JSON, this might form the basis for a powerful and highly usable data distribution and referencing system. Returning to our starting point in this section, this would then need to be provided with a highly ownable and embeddable user interface for manipulating the data held from the point of view of any subcommunity.

5. Ownable Installations
It should be possible to own the physical machines and networks on which the software of individuals and communities runs. Another name for this capability would be “ownable infrastructure” — referring to the entire chain of supports necessary to make a piece of software available in a particular time and place, from the hardware (virtualised or otherwise), the energy and network costs of running it, its connections to sources of data in its community and those of others, the act of installing it itself, paying any licence costs, and ensuring that it stays running stably and free of security vulnerabilities.

We don’t necessarily mean that people should be able to run their own data centres, but this should at least be in theory possible if the ones commercially on offer operate unsuitable policies, e.g. with respect to data privacy, licencing, etc.

Ironically, for web applications, this is an area of ownability in which there has been significantly greater progress than any of the other classes of “ownability” that we consider. The last ten years has seen a vast proliferation of “Something as a Service” platforms by cloud computing vendors, where “Something” can variously be replaced by “Platform”, “Infrastructure”, “Software” or others. Each of these provide models where one may rent standardised parts of the infrastructure needed to host a web application, delegating more or less of the responsibility for it to the provider as needed. These rely on various pieces of orchestration technology, many of which have increasingly solid open source implementation, such as OpenStack and Kubernetes, which automate the processes of spinning up one or more machines of a particular configuration which cooperate to provide a particular function.

This is a highly promising development, although the ecosystem is still in turmoil and is very far from
the state where an ordinary citizen (or expert) could turn up and at the push of a button set up a fully configured incarnation of their own version of a complex platform such as iNaturalist.

One community attempting to meet these aims directly is named hood.ie\textsuperscript{10}. This community is unusual in the technological ecosystem in being wholly oriented towards newcomers, supplying copious and invitingly-formatted documentation aiming to help them get off the ground as quickly as possible with designing and hosting applications which are equally capable of working with data in the cloud as well as locally. This is an open source project which makes use of numerous other open source products (CouchDB for persistence, node.js for running JavaScript on the server) to orchestrate together easy-to-use solutions for non-expert developers. While this is a highly promising development, because of the profound kinds of problems in software structuring we mentioned in our sections on ownable expression, hood.ie is only capable of expressing relatively simple designs convincingly — and once the designs scale beyond this horizon, they may well need reimplementing in more traditional technologies. This onramp is smooth with a shallow slope, but fairly short.

6. Ownable Economies

Citizens should also be able to own the institutions and mechanisms that organise their labour and its products, and which are responsible for supplying them the means of life.

Our running large-scale example, iNaturalist, is a less good fit to our discussion here because of the way it is funded and operated, but it is largely because it has been able to sidestep these issues that it has likely had the space to create such a vibrant and healthy set of communities. It began life as a grant-funded project, and while its current funding model is not clear, it enjoys such an obvious “public good” mission, and its running costs supporting the current rate of about 10 observations per minute are likely to remain modest given the vast commoditisation of cloud computing resources which we’ve seen over the last several years, although naturally this doesn’t cover the much greater cost of maintenance, continued development, outreach and other content creation.

Other communities with a less obvious “public good” mission will have to bite the bullet of issues such as monetization, governance and how ownership of the platform and its technology actually enacts itself in their lives. Many so-called “disruptive” platforms have promised to disintermediate the process of organising labour, but the majority of them, such as Uber, AirBnB, and the like have been found to be punishingly extractive (Wachsmuth et al., 2018). Rather than vehicles for equity, these platforms merely function to channel value away from communities and centralise it in the pockets of distant founders and backers, frequently to be found in Silicon Valley. Other platforms such as YouTube monetize the time, attention and privacy of its users in ways which can be difficult to trace and understand, e.g. the automatically generated YouTube videos targeted at children described by Bridle (2017).

A growing movement of “co-operative platforms” seeks to find ways in which these means of organising labour and work can be truly owned by the participants. The authors are involved in a recently funded project to produce a “Platform Cooperative Development Kit”\textsuperscript{11} to produce some freely ownable infrastructure and blueprints that can assist communities like these.

7. An ideal adaptation scenario

We now imagine ourselves in a world after the revolution, and consider how the communities surrounding our iNaturalist running example might ideally structure their work if we had achieved all of our aims of ownability.

Firstly, following ownable expression, there would be no need for the adapting community to fork the entire code of the core product at all. Part of the necessary substructure for a lens-based design as

\textsuperscript{10}http://hood.ie/

\textsuperscript{11}https://www2.ocadu.ca/news/initiative-co-led-by-idrc-and-the-new-school-receives-1m-google-grant
described in section 3.2.1 would be a *natural alignment* between not only the structure and function of the original design, but also variant designs. The adapting community’s contribution could be structured as a completely self-contained project, and at the time of installation, overlaid onto it using the natural *program addition operator* that such an alignment enables — the nature and requirements of such an operator are described in (Basman et al., 2018). This is currently the most far-fetched of our results.

Secondly, following ownable function and installation, the adapting community would be able to freely make the choice between hosting their own iNaturalist, or attempting to get modifications introduced into the central installation or one of its federated peers.

Thirdly, following ownable data, the adapting community and central community would be able to freely choose amongst themselves the policy on whether updates issued from the adapting community would be fed back to the central community and on what schedule and granularity.

In our post-revolution world, every community would be able to choose from a ready-implemented standardised set of protocols, policies and formats for these purposes.

### 8. Conclusion

We’ve drawn up a set of blueprints for radical progress towards citizen ownership of technological products on multiple fronts. Of these, the ones that relate to the means of expressing and designing software are the most intractable and difficult to achieve, since they involve fundamental reframings of many ideas that have become entrenched in mathematical, metaphysical and computational dialogues for a substantial time. In addition, there is essentially no recognition in mainstream programming language and system design communities of the kind of far-reaching extirpation that will be required. Credible progress in this area will take many decades.

Our ownership aims for actual software installations are less distant. As we mentioned in section 5, there are sufficiently promising developments underway in the infrastructure communities that we could expect such systems to be at the disposal of everyday citizens in a decade or so. However, unless the problems of ownable function and expression were solved, with aligned solutions to ownable data, these systems would be of little use.

The issue of ownable data is also one on which there is a reasonable and growing recognition that there is a substantial problem to be solved, of significant benefit to society, but progress so far is only moderate. This recognition is centred in communities such as data journalism and governmental statistics, and there are even institutions such as the Open Data Institute and the Open Knowledge Labs pursuing a subset of these aims.

It is the fact, as our presentation of a running example has shown, that these aims are intimately interrelated and that lack of progress on any one will hold back the development of several others that makes incremental progress towards this revolution difficult. This is why we have concentrated here on sketching a crude picture of many areas at once, with a rough indication of their relationship, rather than trying at the futile endeavour of proving a prescription of ordered steps for reaching all these goals at the same time.

There is nothing to be derided in this approach, since as Lakatos (1978) points out, genuine incrementalistism is impossible in true science in any case. In his analysis of Newton’s theories of motion and gravitation, he points out that under these, as with any credible general theory, no one part of the real system being explained could ever be effectively separated from distracting effects in order to be an unambiguously validating or refuting phenomenon for the theory. Furthermore, he points out that any observation which is intended to incrementally support an improved theory must first be interpreted with respect to the theory it intends to refute — we must always ‘progress on inconsistent foundations’.

History is replete with the transformative effects that ownable artefacts and infrastructures, such as the printing press, can enact on the largest scales in societies — to return to our revolutionary theme, without
the highly ownable printing presses housed in the Cordeliers district of Paris (Hesse, 1991, Chapter 5),
the inflammatory pamphlets written by Danton, Desmoulins and other revolutionaries could not have begun their work at opposing the extractive power of the French state. We expect our own universal media, once they are endowed with all the properties that ensure their ownability by private citizens, to be capable of similar effects in levelling systemic inequities.

Acknowledgments

This paper emerged in large part from discussions of what revolutionary goals our community should seek with Luke Church, Colin Clark, Clayton Lewis, Mariana Mărăsoiu, Tomas Petricek, Oli Sharpe, and others.

References


Wide, long, or nested data? Reconciling the machine and human viewpoints

Alan Hall, Michel Wermelinger, Tony Hirst
The Open University, UK
{alan.hall, michel.wermelinger, tony.hirst}@open.ac.uk

Santi Phithakkitnukoon
Chiang Mai University, Thailand
santi@eng.cmu.ac.th

Abstract
Data expressed in tables may be re-arranged in various forms, while conveying the same information. This can create a tension when one form is easier to comprehend by a human reader, but another form is more convenient for processing by machine. This problem has received considerable attention for data scientists writing code, but rather less for end user analysts using spreadsheets. We propose a new data model, the “lish”, which supports a spreadsheet-like flexibility of layout, while capturing sufficient structure to facilitate processing. Using a typical example in a prototype editor, we demonstrate how it might help users resolve the tension between the two forms. A user study is in preparation.

1. Introduction
1.1 Background
Data in tabular form are everywhere: government statistics, company accounts, scientific results – to name but three examples. But beyond the fact of being arranged in rows and columns, the term “table” covers a multitude of structures, whose choice of layout can affect both efficiency of processing and ease of human comprehension.

Drawing on relational database theory, Wickham (2014) introduced the concept of “tidy” data as a standard that can be used to facilitate preparing tabular data for use with analytical tools. Mount & Zumel (2017) propose “coordinatized” data, performing a similar role: provided a value can be located in some multi-dimensional space, the underlying model can be agnostic to which form of view the user prefers.

A common pattern is when some of the columns in a table form a time series of observations, each row in the table referring to a single subject. A small example is shown in Figure 1(a). This is the wide form, which is generally the more human-readable but is an example of an “untidy” layout. The long form, where the time series for all subjects are stacked into a single column, is the one more often required by analytical tools, and is shown in Figure 1(b).

![Figure 1 – a small dataset in (a) wide form, above; and in (b) long form, right](image)

The work above relates to code-driven analysis of tables. How might these ideas transfer across to interactive analysis, as performed by end users? The predominant end user tool in this space is the spreadsheet, which has achieved its popularity due to its usability (in particular, support for direct manipulation) and its flexibility (Scaffidi, 2016). But in the first author's experience as a professional analyst, it is rare to see even well-designed spreadsheets in long form. It seems a reasonable assumption that this is not a good cognitive fit for the user's view of the data. Referring to the cognitive dimensions of Green & Petre (1996), there is a stronger closeness of mapping between the user's mental view of the data structure and the wide form than the long one. This may arise from the
enhanced visibility of series that are logically continuous or juxtaposed when the two-dimensional space is effectively utilised. The secondary notation dimension is also relevant, in that a user can choose to use spacing, shading, gridlines, etc. to help visualise the structure.

Perhaps not surprisingly, previous work on making the spreadsheet “tidier” has taken the direction of making it behave more like a relational database. Bakke & Karger (2016) describe a spreadsheet-like interactive query builder for a backend database, while Chang & Myers (2016) describe a similar tool which takes JSON as input. Cervesato (2007) and Hawkins et al. (2014) both extend the spreadsheet itself with the relation as a native object type, while Mangano et al. (2011) describe a hybrid model in which relational and freeform data can coexist.

These approaches require the user to shift their mindset from arrangements of cells, to entities with attributes. Even the pivot table (and its more recent counterpart, the unpivot command) forces the user to this kind of dual view of their data. We are exploring whether an alternative data representation might allow the user to remain “untidy”, but still provide the machine with sufficient information to assemble the long form implicitly, behind the scenes. Hence the user may retain the flexibility of layout that is a strength of the spreadsheet, but be relieved of the burden (both mental and mechanical) of maintaining and switching between dual copies of the data, the wide and the long. The machine for its part is able to use the deduced structure to facilitate onward processing and future maintenance of the model.

---

**Figure 2 – the building blocks of the “lish” data model**
1.2 The “lish” data model
In an earlier paper (Hall et al., 2017) we have introduced the “lish” as an alternative model for spreadsheet-like data. The lish is a list-based model whose main characteristics are shown diagrammatically in Figure 2. In brief, a lish replaces the usual grid of cells with a list. Each element of a lish can be either a single cell or a further lish, so nesting is possible to any depth. This enables the lish to capture grouped and hierarchical structures more expressively than a grid, but a one-dimensional list is a poor abstraction of a two-dimensional table. So in order to express table- and array-like behaviours, the first element of every lish has a privileged status: it forms a “template”, defining a minimum structure with which subsequent elements must conform. For example, a template that is a list of five cells constrains further elements of that lish to be also lists of five cells. Such a lish could represent a table with five columns and any number of rows. By applying the template rule recursively, we can represent more elaborate or higher dimensional structures. In addition, we developed a typesetting algorithm which ensures that elements associated with a common template are aligned in an intuitive table-like way.

Here, we extend our earlier work by introducing calculations (after the manner of spreadsheet formulae) to the lish, with particular reference to the wide vs. long problem. Our goal is to bridge the cognitive gap between a visual layout that accords with the user’s mental model of the data, and a machine-friendly layout that will facilitate calculation.

2. Case study
2.1 The scenario
We will illustrate an application of the lish using a small fictitious case study. A local chain of retail outlets has four branches within a town, denoted here simply as North, South, East and West. For budget monitoring purposes they have collected data on monthly footfall and revenue at each outlet. The chain is closed on Sundays, and the number of trading days per month has also been recorded. Using these data, they would like to perform a series of ad hoc calculations, e.g. for total monthly revenue, and revenue per trading day.

The data as entered into our prototype lish editor are shown in Figure 3. This layout is “untidy” – not only is it in wide form, but it has some freeform notes along the bottom of the table which the analyst decided to add, commenting on unusual circumstances in certain months. The grey shaded cells are our templates: they are either the first element in a lish, or part of a sublist that is itself a template. Figure 4 shows the same data in long form; in a normalised database, month would be a foreign key in this table, referring to a separate small table holding the trading days per month information.

2.2 A grouped aggregation
Let us now consider the task of calculating the total revenue across the four branches, for each month. If using a code-based tool, we should prefer the data to be in the form of Figure 4. The calculation would then proceed e.g. using a GROUP BY query in SQL. In a spreadsheet, we could accommodate the same layout using SUMIF, but might prefer the wide layout of Figure 3, which plays better to the directness supported by the spreadsheet formula model. Instead of working at the level of the whole table, the user would enter in the first cell of the column a SUM formula referring to the individual January revenues for the four outlets, and then copy it down the column.

There is a problem with the wide spreadsheet layout, however: although it makes the form of the data more comprehensible, and formulae more direct, the approach just mentioned would not scale well to the addition of extra retail outlets. In general, spreadsheet formulae involving cells that are logically related but not physically adjacent on the sheet are more fiddly to construct and error-prone than those involving the selection of a single contiguous range.

To produce the monthly totals, we must specify two things: which cells are to be summed (here, all of the revenue cells), and across what dimensions to sum them (here, we want the row sums – other options might be the column sums, or the grand total). The lish can help on both counts.

First, the editor supports a cell selection method whereby navigating to any template cell implicitly selects all those cells for which it is a template, recursively. Go to a column label, and you implicitly
select the column; go to the top left cell in a table, and you select the whole table; go to the very first cell in the outermost lish, and you “select all”. The lish in Figure 3 has been arranged such that a single sublist within the main table represents the four smaller tables for the individual branch outlets, labelled North, South, East and West respectively. The $7 \times 3$ region of grey cells beginning with “location”, near the left hand side, is the template for this sublist; it could be visualised as the base plane upon which identically shaped tables, one for each outlet, are to be stacked. So in the figure, selecting the column heading for “revenue” has selected that column, which in turn has selected the equivalent columns in all of the outlets. Although we have “wide” columns, this selection has unfolded a “long” column that was hidden in plain sight in their midst.

![Figure 3 – The retail outlet data in wide form.](image)

![Figure 4 – The same data as Figure 3, but in long form (first few cases only).](image)
Second, the editor can deduce from the structure, in just the same way, that the `total_revenue` cell labels a single column. So it “knows” that any summary statistics requested from this cell are to be calculated row-wise. The long column selected in the previous paragraph retains the memory of its fold points, so can be segmented correctly to produce the required monthly totals.

The formula for total revenue is `sum($location.revenue)` and is defined only once, in the head of the `total_revenue` column. From here, it populates the entire column. The dollar in the syntax denotes a labelled location in the lish (as opposed to a function name, like `sum`). The dot, in the expression `location.revenue`, avoids potential ambiguity by designating the intended `revenue` label to be the one within the `location` lish. That expression could have been typed verbatim, but was filled in automatically by the machine: the editor supports a spreadsheet-like interactive mode for building formulae, in which the user may navigate to a cell in the midst of formula construction and have that cell’s label inserted. So both components of the specification – what to sum, and across what dimension to sum it – have been specified entirely visually, and in each case by pointing to a single cell. We don’t need an explicit `GROUP BY` as with the long form. Nor do we have to reference individually all the columns to be summed, as with the wide form.

2.3 A binary operation
The lish in Figure 3 contains another calculated column (at the far right hand side) in which the revenue per day has been calculated for each month. The formula for this is `$total_revenue / $trading_days`. Both the numerator and denominator are one-dimensional lists of the same length, so the division is carried out pairwise between their corresponding elements. The result is a third list of the same length.

Now let us add a slightly more complicated calculation. Suppose we would like revenue per trading day, as before, but this time broken down by outlet as well as by month. To achieve this, the user first created an extra column in the original 7 x 3 sublist that is the template for the individual outlets. Since this sublist is a template, the editor automatically added a similar column for each individual outlet. Then, the user created a formula, `$revenue / $trading_days`, at the head of the new column in the template (the `location` qualifier for `revenue` is not needed this time, as unlike in subsection 2.2 the `revenue` column is local to the sublist that contains the formula). An excerpt from the resulting lish is shown in Figure 5.

Once again, pairwise divisions have been carried out between the elements of `revenue` and the elements of `trading_days`, but this time there is not a one-to-one correspondence between the two. As in the previous subsection, the machine “knows” from the nested structure that, for instance, the single value of 25 trading days in April, occurring in the denominator, is the counterpart to each of the four revenue values (one per location) for April, where they occur in the numerator. In the next section, we provide an overview of how such deductions are made.

3. A brief sketch of lish calculus
In the previous section, we saw examples of the machine using the structure of nested lists and templates which make up a particular lish to reason intelligently about the calculations required. The results accord intuitively with our notion of tabular structures, even though the primitive operations concerned are not defined on those structures: aggregation functions (like summation) are defined on one-dimensional arrays, and binary operators (like division) are defined on pairs of scalars.
Lish calculus is the name we give to our collection of algorithms that extend the definition of these operations to the lish. It draws heavily upon the notion of vectorised arithmetic, as defined by the R programming language (R Core Team, 2018). That language is itself based upon the usual mathematical definitions of vector and matrix operations, but with the novel addition of a “recycling rule”, which provides greater flexibility by accommodating operands that may be vectors of different lengths.

In order to operate upon lishes, we modify vectorised arithmetic in two main ways. The first is fairly trivial: since the first element of each lish is a template and does not contain ordinary data, it must be omitted from any arithmetic. For example, if we sum a lish, we do not attempt to include the first element within the sum. The second modification is more complicated: we need a version of the recycling rule that will accommodate operands that may be not just of different lengths, but different depths as well. For example in subsection 2.3, `trading_days` was a one dimensional list of cells, whereas `revenue` was a list of lists of cells. The details of binary operators as defined over lishes are quite intricate, due to their recursive nature, and are beyond the scope of this paper. But the principle that underpins them is very simple: we perform pairwise matching between those sublists that were derived from the same template list. In our implementation, we cache the template with each sublist to improve the efficiency of this matching. A similar principle applies to aggregation functions, like `sum`: we compare those templates that appear within the lish being summed to those within the lish that is destined to hold the result. Templates that appear in the former but not in the latter identify which dimensions are to be summed over.

4. Discussion

The case study showed that the lish embodies more of the structure of the data than a flat grid, and that the machine can use this structure in ways that may help the user. But there is an obvious objection: when entering the data in lish form, did the user not have to do some extra work up front in order to define the structure in the first place? If so, perhaps on balance they are no better off than entering the data in a spreadsheet and performing the wide to long transformation explicitly. We are planning a user study which should provide some empirical evidence to this question. In the meantime, we make here a common sense argument based on relative costs and benefits.

Starting to build a model as a lish has some costs compared to building the same model in a spreadsheet. A blank lish is just a one-dimensional sequence of empty cells, in which tables only appear once the user starts enclosing sublists from among those cells. Our critical assumption is that the nested structure of the lish mirrors the way the user visualises the natural hierarchies in their data (closeness of mapping, again), so that forming a sublist requires minimal mental effort, as well as minimal mechanical effort (one key-press).

On the credit side, the lish repays the user with some early gratification once a basic structure is in place. We saw in the case study that formulae can be fewer and simpler than in a spreadsheet, and instantly populate all their relevant cells. A similar benefit occurs when cells are inserted or deleted in a template: all dependent tables are grown or shrunk to match accordingly. Just as the time-proven spreadsheet model gives the user a feeling of direct control by instantly updating `values`, so the lish complements that desirable attribute by instantly updating `structure`.

As an alternative to having the user explicitly lay out the structure, by grouping cells into sublists, we might consider inferring it by assuming certain spreadsheet conventions (such as column labels forming the first row of a table). This approach has been successfully applied before: e.g. by Hermans et al. (2011) to mapping data flow at varying granularity; by Hodnigg & Pinzger (2015), to parsing a worksheet into cognitive units; and by Kankuzi (2017), to abstracting formula calculations in narrative form. It has the advantage that it can be applied to a conventional spreadsheet, but would appear to be more challenging with higher dimensional structures such as those occurring in our case study. There is also the distinction that explicit structures like the lish produce a clearly deterministic result, whereas inference relies to some extent on the machine making a correct interpretation of flexible conventions.
A limitation of the lish is that it has no concept of a foreign key relationship. Since it is a nested structure it can express relationships of a form where one type of object “owns” a fixed or variable number of some other type of object. Hence, it can represent a limited class of 1:n relationships without the need for a foreign key as such. But it is not intended to be a replacement for the relational model itself. Its domain of application lies rather in a middle ground, as exemplified by our retail outlets case study, where multi-dimensional data or families of similar tables present scalability problems for the spreadsheet, but a relational database would feel over-engineered.

5. Conclusions and future work
We have seen that there can be a tension between data representations that are easily comprehended by humans and those that are more conveniently processed by machine. A case in point is the choice between “wide” and “long” tabular data, often occurring where a time series is involved.

The “lish” provides a data model that supports the more human-oriented form for visualisation, while enabling the machine-oriented form to be accessed readily for calculation. We have shown in our case study of retail outlet data how an existing model may be expressed in lish form, and how the use of template cells and implicit data selections can assist the end user in constructing an analysis.

Our next step will be to conduct a user study to evaluate these ideas. First we will seek to verify our assumption that users comprehend tabular data in the way that we have assumed, and that the lish representation of “chunks within chunks” does indeed map closely to the user’s mental model of a sequence of tables. We will also seek to evaluate the usability aspect. In particular, we would like to assess users’ perceptions of the relative costs and benefits of constructing a model in lish form – and if necessary, what we might do to make this balance more favourable.

References


R Core Team (2018) R Language Definition v 3.5.0, section 3.3, “Elementary arithmetic operations”. Available at: https://cran.r-project.org/doc/manuals/r-release/R-lang.html


Investigating multimodal affect sensing in an Affective Tutoring System using unobtrusive sensors

Hua Leong Fwa
Department of Computer Science
Newcastle University
h.l.fwa1@newcastle.ac.uk

Lindsay Marshall
Department of Computer Science
Newcastle University
Lindsay.marshall@newcastle.ac.uk

Abstract
Affect inextricably plays a critical role in the learning process. In this study, we investigate the multimodal fusion of facial, keystrokes, mouse clicks, head posture and contextual features for the detection of student’s frustration in an Affective Tutoring System. The results (AUC=0.64) demonstrated empirically that a multimodal approach offers higher accuracy and better robustness as compared to a unimodal approach. In addition, the inclusion of keystrokes and mouse clicks makes up for the detection gap where video based sensing modes (facial and head postures) are not available. The findings in this paper will dovetail to our end research objective of optimizing the learning of students by adapting empathetically or tailoring to their affective states.

1. Introduction
The critical role that emotion plays in learning is affirmed in a number of studies (Kort, Reilly, & Picard, 2001; Pekrun, Goetz, Titz, & Perry, 2002). Pekrun et al. (2002) conducted a series of quantitative and qualitative studies which concluded that positive affect e.g. enjoyment of learning affect achievement positively by strengthening motivation and enhancing flexible learning whereas negative affect such as anxiety and frustration erodes motivation and draws attention away from the task, resulting in shallow learning. These affirmed the role that affect plays in the learning process as well as the need to mitigate the effects of negative affect such as frustration.

Intelligent Tutoring Systems (ITSs) are built with the objective of providing learners with the benefit of one to one tutoring automatically and cost effectively by tailoring instructions to individual learning needs (Psotka, Massey, & Mutter, 1988). It acts as a personal training assistant that continually assesses one’s knowledge through interactions with the system and builds a personalized model of one’s acquired knowledge for the provision of tailored instructions or assistance in the form of hints or demonstrations when one seems to require help to move on. Some studies have in fact shown that ITSs outperform traditional classroom instruction in some domains (Anderson, Corbett, Koedinger, & Pelletier, 1995). However, for most domains, ITSs still underperform one-to-one tutoring. This has been attributed to the lack of emotional awareness in ITSs (Alexander, 2004; Picard, 1997).

Affective Tutoring Systems (ATSs) are Intelligent Tutoring Systems (ITSs) that adapt their tutoring responses based on the sensed emotions of the students, resulting in enhanced tutoring outcomes. The first step to building an ATS is to equip it with the ability to sense the emotions of the tutee. There are various affect sensing techniques and among the many, sentic modulation (Picard, 1997) is the technique which shows much promise in dynamic affect sensing. Sentic modulation refers to the physical assessment of a person’s emotional changes via sensors such as cameras, microphones and wearable devices that register subtle physical modulation produced by emotional states.

Most previous studies on the use of sentic modulation have used a single sensor (unimodal) for affect sensing. Humans, on the other hand express our affect in multiple channels e.g. facial expressions, body postures and vocal intonations which thus leads to the belief that the use of multiple modalities for affect detection would more closely emulate human affect expression. Multimodal affect sensing is hypothesized to be superior to unimodal affect sensing as it is commonly believed that the multiple sensors complement one another. It also offers the affordance of data missing in some of the modalities as other modalities can make up for the missing data albeit at a degraded performance.

As compared to unimodal affect detection, the use of multiple modalities involves more technical complexities and issues. Baltrušaitis, Ahuja, and Morency (2018) aptly summarizes the technicalities
of multimodal affect detection into the 5 categories – Representation, Translation, Alignment, Fusion and Co-learning. Representation refers to the representation of heterogeneous multimodal data for the exploitation of complementarity and redundancies in multiple modalities. Translation refers to the mapping of data from one modality to another. Alignment refers to the identification of relations between elements from two or more modalities. Fusion refers to the joining of information from the modalities to perform a prediction while Co-learning relates to how knowledge learned on one modality can be transferred to a computational model trained on another modality.

Keyboards and mouse are standard input devices attached to every computer in a computer lab. Web cameras though not as common, are cheap and can easily be setup for use. Conceivably, these would qualify as economical commodity devices that are unobtrusive and thus suited for capturing the behavioural attributes of users inconspicuously.

In this study, we detail the techniques that we adopted to address some of the issues relating to multimodal affect sensing and to investigate into whether a multimodal and unobtrusive array of keystrokes, mouse clicks, facial features, head pose and contextual logs enhances the accuracy of affect sensing as compared to unimodal means.

2. Methodology

2.1. Participants

This work was compiled from data gathered in the trial conducted in Nanyang Polytechnic, Singapore in the year 2014 and 2015. The study was conducted in computer labs where 22 students were enrolled to work on programming exercises for an average period of 72 minutes in a Java programming tutoring software. Before the start of the session, students were requested to fill in a form granting consent to participate in the study. The students were also briefed on the objectives of the study and what was required of them in the trial. They were also guided on the various functions within the tutoring system.

2.2. Tutoring System

The students were provided with a total of 12 exercises to be completed within the tutoring software. Each exercise was preloaded with a set of Java codes with missing lines in between and students will have to fill in the missing lines of code to complete the exercise. Within each exercise page, students can click on a “Submit code” button to submit the code for compilation. They can then check the output window for the compilation output or errors. The compilation output will be verified against the required output when the student clicks on the “Check answer” button and will be marked as completed if the correct output is obtained.

The students’ actions, keystrokes and the respective time stamps in which each action or keystrokes occurred within the tutoring software were recorded. Some examples of the actions that were logged include the start and end time of each exercise, the time stamp of each “Submit code” action and the time when the exercise was completed. Copying and pasting of code from other web pages outside of the tutoring system was a common behaviour among novice programmers and that was captured in the keystroke logs as well.

2.3. Annotation of frustration

The face and screen video recordings were used for a retrospective annotation of frustration by two lecturer observers with an average teaching experience of five years. This retrospective observation technique was employed by a number of prior studies (Cetintas, Si, Xin, & Hord, 2010; Graesser et al., 2006; Mcquiggan, Lee, & Lester, 2007). Both the facial and screen video recordings were used for the annotation of frustration as it is difficult to ascertain whether the student is frustrated from facial expressions alone. The addition of screen video enhances the annotation accuracy by providing an additional source of information into the cause of the student’s frustration. For example, if the observer observed signs of frustration from the facial video but was not so sure, the observer could then confirm that the student was indeed frustrated (from the screen video) if for example, he or she observed that the student had been trying to rectify a compilation error for quite some time without success.
Some examples of the frustration behaviours noted in the session include use of expletives, long sighing, excessive gesturing and roughly ruffling through hair while visibly distressed. The observers find that these behaviours are usually accompanied with the encountering of compilation errors or being stuck in a particular point in the exercise for a period of time (which can be observed from the students’ screen videos). The observations were recorded with a time stamp which was used for synchronizing with the captured contextual and keystroke logs.

The average length of each student’s video segment is 72.3 minutes (with a standard deviation of 10 minutes) making up a total of n=9502 instances (overlapping time window slices using the sliding window mechanism) in the data set. There was an average of 15 instances of frustration (with a standard deviation of 6) noted per student’s video segment.

2.4. Features
The facial features are captured from web cameras installed on top of the desktop monitors. The iMotions FACET software (iMotions, 2017), a commercial version of the Computer Expression Recognition Tool (CERT) was used for extracting likelihood estimates of 17 Action Units (AUs) from the captured videos. These 17 AUs denote facial muscles movements of the brow, eye lid, nose and lip. From these 17 AUs’ likelihood estimates, we derived additional features by calculating the median, maximum and standard deviation of each, making a total of 51 features. These 51 features are temporally aligned with frustration observation in the 30 seconds time window.

The list of contextual features includes the number of exercises completed, the number of submissions for compilation of the exercise, the number of switches between the exercises, exercise worked on, exercise duration and the number of errors encountered. The number of exercises completed denotes the number of exercises that the student has completed. The number of submissions for compilation denotes the number of times the student has submitted the code for compilation for the designated exercise. During the course of the study, we observed that students click on the submit for compilation button several times within a span of two seconds, thinking that more clicks will help to speed up the compilation time. Thus, to prevent duplicate counting, all submissions for compilation logs (by the same student for the same exercise) time-stamped within duration of two seconds are only counted as one submission for compilation.

The duration or flight time of the key is the duration from the time a key is depressed to the time when the next key is depressed. The keystroke log files on the server for all the students are consolidated and processed using a batch program to calculate the mean, median and frequency of the different groups of keys (e.g. alphanumeric keys, navigation keys, backspace keys e.t.c.). The duration between keys refers to the flight time (the duration from the time one key is depressed to the time the next key is depressed) and the wait duration refers to the duration in which no key was depressed.

Mouse clicks were captured only when the students were working on the exercises. A JavaScript function running on the client end captures and sends the raw mouse clicks information to the server. At the server end, a program processes the raw mouse clicks information which consists of the coordinates of the mouse click, characteristics of the mouse click (single click or double clicks) and time stamp denoting the time of depression of the mouse click. This processed information is then written to a log file on the server. The logged mouse clicks files on the server are then consolidated and processed using a batch program to derive the total number of clicks, number of clicks less than 2 seconds and the number of double clicks.

The head posture features were captured from web cameras installed on the top of the desktop monitors. An eye tracking software from xLabs (xlabsgaze.com) was used for extracting raw head pose information such as horizontal and vertical head position, head roll, pitch and yaw from the captured videos. These were further processed to derive the median, standard deviation and maximum position and the pitch, roll and yaw velocities. The pitch, roll and yaw velocities were derived from the differences of the current pitch, roll and yaw values from the pitch, roll and yaw values of the previous second.

The facial, head pose, contextual, keystrokes and mouse clicks features were lastly combined into a single channel feature vector that is passed to the classifiers for classification.
2.4. Synchronization of features with annotations

The students’ facial, head postures, contextual, keystrokes and mouse logs were aggregated into features using a sliding window size of 30 seconds with an overlap of two-thirds of the window size. If the time at which frustration was observed falls in the overlap area of 2 consecutive window slices, both window slices would be annotated as slices in which the student experiences frustration. Alternatively, if the time at which frustration was observed falls outside the overlap area, only the time window slice in which it occurred in will be annotated as the slice in which the student experienced frustration. This is illustrated in Figure 1.

![Figure 1 - Overlapping time window slices to synchronize features with annotations](image.png)

2.5. Features Selection and Classification

The frustration detection classification models in this study were built using the 3 separate channels (facial channel, contextual, keystrokes and mouse clicks channel and head pose channel), feature fusion (2 channels and 3 channels) and decision fusion (using max and mean vote), making up a total of 7 models.
The 3 base channels are the channels that combine the contextual, keystrokes and mouse clicks, the facial channel and lastly the head pose channel. The feature fusion model combines the features from the 3 base channels into a combined feature vector for feeding into the classifiers. The decision fusion model uses the combination of the decision probabilities from the 3 base classifiers (built using the features of the 3 base channels’ features respectively) for a decision on the final classification output. The final classification output for the decision fusion model can then be determined for instance, by taking the highest of the decision probabilities among the 3 classifiers.

The 3 classifiers used in the models were Random Forests (RF) (Breiman, 2001), Logistic Regression (LR) and Naïve Bayes (NB). The features are extracted from the facial, head posture, contextual, keystrokes and mouse logs. In total, 51 features are extracted from the facial channel, 40 features from the combined keystrokes, mouse clicks and contextual channel and 11 features from the head posture channel. For the feature fused model, 30, 40, 50 and 60 features selected using the RELIEF-F (Kononenko, 1994) feature selection algorithm are fed into the RF, LR and NB classifiers.

RELIEF-F algorithm first selects a random instance and determines the instances that are closest to the selected random instance using the Manhattan distance. The weights to features are then updated through reducing them by the features’ absolute difference for instances that are close to each other and of the same class type and increasing them by the features’ Euclidean distance for instances that are close to each other and of a different class type. The features are then ranked by weights and the top ranked n features are then selected for use in the classification.

The cross validation, feature selection and classification workflow for feature fusion models is shown in Figure 2.

2.5. Student-level nested cross validation

To ensure the generalizability of the classification models, all the models were tested using student-level nested cross validation. In the outer loop of the cross validation, the data set for a randomly selected 66.7% of the students were used as the training set with the data set for the remaining 33.3% of the students as the test set. In the inner loop, a further 66.7% of the student’s data within the outer...
training set (44.5% of the whole data set) were then used for feature selection. The feature selection results were then averaged over 10 runs of the inner loop. In addition, lasso regularization was applied for logistic regression with the lambda regularization value determined through 5-fold cross validation. The classification results for each of the models were averaged over 30 runs of the outer loop to derive the final classification results.

The Area under the Receiver Operator Characteristic Curve (AUC) is used as a performance measure to compare between the various models as it is useful for domains with skewed class distribution and unequal classification error costs (Fawcett, 2006). It is preferred here as compared to accuracy as a performance measure because instances of frustration are rare as compared to non-frustration. A naive classification model that always predict non-frustration for all test instances will have a high accuracy measure but low AUC measure as it does not discriminate instances of frustration from non-frustration.

### 3. Results

In this section, the classification results of the models for discriminating between instances of frustration from non-frustration for the different channels are reported. The AUCs for the facial channel (FC), head pose channel (HPC) and keystrokes, mouse clicks and contextual channel (KMC) using the Random Forest, Logistic Regression and Naive Bayes classifiers are shown in Table 1.

<table>
<thead>
<tr>
<th>Channels</th>
<th>Classifiers</th>
<th>No. of features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random Forest</td>
<td>Logistic Regression</td>
</tr>
<tr>
<td>Facial (FC)</td>
<td>0.552</td>
<td><strong>0.58</strong></td>
</tr>
<tr>
<td>Head Pose (HPC)</td>
<td>0.51</td>
<td>0.542</td>
</tr>
<tr>
<td>Keystrokes, Mouse clicks and Contextual (KMC)</td>
<td>0.539</td>
<td><strong>0.575</strong></td>
</tr>
</tbody>
</table>

Table 1: Classification results for various channels by classifiers

From the results, the facial channel offers the best performance (AUC=0.58) followed by the keystrokes, mouse clicks and contextual logs channel (AUC= 0.575). 25 facial features, 5 head pose features and 20 combined keystrokes, mouse clicks and contextual features are extracted for FC, HPC and KMC respectively using the RELIEF-F feature selection algorithm. The classifiers for each of the 3 channels are better than the random model with AUC=0.5, thus providing evidence that each of the 3 classifiers can discriminate between instances of frustration better than chance. It can also be seen from Table 1 that the logistic regression classifier offers the best performance among the 3 classifiers for FC and KMC.

The classification results for the various fusion models that combine the channels are shown in Table 2. The 3 channels feature fusion model combines the features for the FC, HPC and KMC into a large feature vector for classification. The 2 channels feature fusion model combines the features for the FC and HPC into a large feature vector for classification. A range of features from 30, 40, 50 and 60 features are extracted using the RELIEF-F algorithm for the feature fusion models. For the decision fusion (max) model, the maximum of the decision probabilities from the base classifiers is used as the final decision output of the model. For the decision fusion (sum) model, the decision probabilities from the base classifiers are averaged to determine the final decision output of the model.

<table>
<thead>
<tr>
<th>Fusion Models</th>
<th>No. of features</th>
<th>AUC (Logistic Regression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 channels feature fusion</td>
<td>30</td>
<td><strong>0.636</strong></td>
</tr>
<tr>
<td>2 channels feature fusion</td>
<td>30</td>
<td>0.631</td>
</tr>
<tr>
<td>3 channels feature fusion</td>
<td>40</td>
<td>0.621</td>
</tr>
<tr>
<td>2 channels feature fusion</td>
<td>40</td>
<td>0.607</td>
</tr>
<tr>
<td>3 channels feature fusion</td>
<td>50</td>
<td>0.582</td>
</tr>
</tbody>
</table>
2 channels feature fusion & 50 & 0.58 \\
3 channels feature fusion & 60 & 0.568 \\
2 channels feature fusion & 60 & 0.568 \\
Decision Fusion (Max) & FC:25, HPC:5, KMC:20 & 0.578 \\
Decision Fusion (Average) & FC:25, HPC:5, KMC:20 & 0.588 \\

| Table 2 – Classification results for the various fusion models using Logistic Regression |

The results show that the 3 channels feature fusion model using 30 features has the best performance (AUC=0.636) among the fusion models. The AUC of 0.636 of the feature fusion model is 9.7% higher than the best unimodal channel (facial channel) with an AUC of 0.58, verifying that multimodal fusion leads to higher detection accuracy over unimodal model. In general, the feature fusion models perform better than the decision fusion models (for those feature fused models with 50 or lesser features).

The AUC of the 2 channels feature fusion model which combines FC and HPC is only slightly lower than that of the 3 channels feature fusion model. This shows that the inclusion of keystrokes, mouse clicks and contextual channel in the 3 channel model only slightly enhanced the classification performance compared to that of the 2 channel model that includes only facial and head pose channel features. However, it is still relevant to include keystroke, mouse clicks and contextual features in the fusion model as the facial and head pose features are unavailable for an average of 17% of the total sessions across all students. If we deploy this tutoring system in an actual classroom environment where lighting and occlusion issues are more prevalent, the availability of the facial and head pose channels will be further reduced. Thus, the addition of keystrokes, mouse clicks and contextual channel features complements the affect detection using facial and head pose channel features.

3. Conclusion

The main goal of this study is to explore automated techniques for the detection of frustration in a naturalistic learning environment. With adequate detection of frustration on a moment by moment basis, hints and tutorial supports can be provided to the students to overcome learning barriers and alleviate their frustration so as to sustain their engagement in learning.

In this study, we have explored the use of unobtrusive sensors in affect detection for the context of a tutoring system that tutors students in programming. More specifically, we have established the viability of using keystrokes, mouse clicks and contextual logs for the detection of frustration on a level of granularity that is adequate for timely remedial intervention.

Keystrokes and mouse clicks are traditionally used in computer security domain for authentication and user identification purposes and are rarely used for affect detection and thus, the significance of the use of keystrokes and mouse clicks for affect detection in this study. Importantly, we are also confident of the generalizability of our results owing to the use of student-level nested-cross validation for validating the models.

Another area of significance is in multimodal affect detection – the fusing of multiple sensing modes outputs. It is a fact that human emotion is expressed in various channels e.g. facial, vocal and bodily expressions but implementation of a multimodal affect detection system is still rare in occurrence (Jaimes & Sebe, 2007). In this study, a multimodal system of affect detection using keystrokes, mouse clicks, contextual logs, facial and head postures combined using both feature fusion and decision fusion techniques is proposed and implemented. It is further verified that a multimodal fusion of the proposed sensors does outperform the best unimodal channel (the facial channel). Although the features that contribute most to the accuracy of the multimodal model are the ones that are derived from head postures and facial channels, the keystrokes and mouse clicks do make up for the periods of detection gaps where both the head postures and facial features are not available.

An extension of this study will be to implement the above described affect detection technique in a programming tutoring system. Augmenting the tutoring system with the ability to infer the affect of
students is the first step in the construction of an ATS. The challenge consequent to this will be to design the ATS such that it can respond appropriately to the detected emotions of the students to increase engagement, task persistence and sustain their motivation to learn. This will entail the tailoring of various affect-driven and pedagogical focused measures e.g. hints and empathetic supports to enhance the tutoring effectiveness of the ATS, thus effectively closing the loop of affect detection and intervention.

4. References
Explicit Direct Instruction in Programming Education

Felienne Hermans and Marileen Smit
Delft University of Technology
{f.f.j.hermans, m.i.e.smit}@tudelft.nl

Abstract
In education, there is and has always been debate about how to teach. One of these debates centers
around the role of the teacher: should their role be minimal, allowing students to find and classify
knowledge independently, or should the teacher be in charge of what happens in the classroom, explain-
ing students all they need to know? These forms of teaching are also called exploratory learning and
direct instruction respectively. While the debate is not settled, more and more evidence is presented
by researchers that explicit direct instruction is more effective than exploratory learning in teaching
mathematics and science. These findings raise the question whether that might be true for program-
ing education too. This is especially of interest since programming education is deeply rooted in the
constructionist philosophy, leading many programmers to follow exploratory learning methods, often
without being aware of it. This paper outlines this history of programming education and additional
beliefs in programming that lead to the prevalence of exploratory forms of teaching. We subsequently
explain the didactic principles of direct instruction, explore them in the context of programming, and
hypothesize how it might look like for programming.

1. Introduction
Programming education is in fashion: in many countries around the world, programming is mandatory
now in the UK, Australia and the US. This is of course, wonderful news for the lovers of programming!
In the future, more children will be able to use their power to make computers do their bidding. How-
ever, programming is not an easy skill to learn, as evidenced, for example, by high dropout rates in
introductory programming courses. While we do not have any numbers on the success rates of courses
aimed at younger children, it is likely that there too learners are struggling.

In this paper we examine the form of teaching programming. In other fields like mathematics and
language, recent results show that direct instruction is more effective, and benefits weaker students
more.

In this paper we examine what programming believes about teaching. What are our didactic and philo-
sophical beliefs and where do they come from. We present a brief history of the didactic beliefs of
programming education, and explain why so far they have mainly leaned towards exploratory learning.

The paper also presents explicit direct instruction, since many programmers are not aware of it and might
be hesitant for that reason. We close the paper with a detailed look into how we currently implement
didactic principles of direct instruction and how we could benefit from its insights.

2. Constructionism and programming
When we program and when we teach programming, we also bring our beliefs about programming into
the classroom. Contrary to educational scientists, psychologist and the like however, computer scientists
are often unaware of their didactic beliefs, since we are not used to discussing those. There is however
one didactic philosophy that many programmers believe in, and that is constructionism. In this section
we explain why that is and what the implications of this are.

2.1. LOGO and its constructionist history
Programming education aimed at young children has a long history, starting in the sixties most notable
by Seymour Papert and his work on the LOGO programming language. While many programmers are
aware of LOGO, and have even learned to program in it, fewer people are informed about the didactic
and philosophical ideas behind LOGO. Papert was a multidisciplinary research with a BSc in philosophy
and a PhD in mathematics, and he worked with a variety of researchers.

One of them was Jean Piaget, one of the most well-known psychologist of the last century, who pio-
neered the idea of constructivism (which is not constructionism, we will come to that in a bit!). The
basic, simplified idea of constructivism is that children build their own view of the world, among other
activities through play. Earlier researchers has seen playful exploration of children as unrelated to learn-
ing, but Piaget argued that children assimilate knowledge and then change their mental models of the
world in a process called accommodation. Piaget aimed to describe how children built knowledge, nit
not how to teach. This is where Papert comes in. He worked with Piaget for years and Piaget once said
that Papers was the one “who understood his ideas best”. Papers coined the idea of constructionism,
which is a didactic method based on constructivism. Constructionism advocates students exploring the
world around them, in a process often called discovery learning. Students use their existing knowledge
to acquire more knowledge via assimilation and accommodation as Piaget described those. The role of
the teacher is more of a coach than a lecturer providing step-by-step guidance. Students often engage in
project-based learning, making connections between different ideas and scientific fields rather than relying
on predefined courses and boundaries of fields (Alesandrini & Larson, 2002). From this philosophy
also LOGO was born, in his book Mindstorms (Papert, 1980), Papert describes the great ease with which
young children learn French if they live in France. With LOGO, Papers wanted to create a mathland
in which it is similar easy to learn mathematics and programming. While we do not often discuss this
didactic basics of programming education, many programmers do believe this philosophy, thinking that if
we just let children program, they will learn how a programming language works with ease. And while
modern programming languages for children, for example Scratch, were not directly created by Papert,
it was created by the same research group at MIT, and once you know about the constructionist beliefs
Papert used to design LOGO, you see the same in the Scratch interface and the accompanying books
and lessons. They too emphasize trying and exploring over detailed explanations of how things work in
detail.

2.2. No collective memory of programming lessons
A second reason that there is massive buy-in into the constructionist mindset is a different one. Most
professional programmers today are in the thirties and forties. That means that, when they learned to
program, there were no programming courses, or websites. Most programmers taught themselves pro-
gramming to a certain extent, and having taught oneself is a source of great pride for many programmers.
Some job ads even specifically search for new employees that have been "programming since they were
12". As such, the programming community really lacks a collective memory of how one would give
programming lessons. In many cases people that teach programming are programmers themselves, and
thus they repeat the childhood experience of those teaching, resulting in very exploratory lessons, or, as
was common in the eighties, by giving children large programs to copy, leaving the exploration up to
the children.

And while it is great that there are children that can teach themselves, it does not mean that all children
can learn programming that way. Like there certainly are some children that teach themselves to write
or to bike, the large majority of children need some amount of help in learning various skills.

2.3. Programming skill is innate
In addition to the above, there are additional beliefs that fit the constructionist way of thinking. One
of these is the widespread belief that not everyone is born for programming. Many people believe that
programming is a skill that is not for everyone. While this seems too obvious to mention, it differs from
general beliefs about reading and writing. Almost everyone believes that everyone with normal intel-
ligence can learn how to read and write, even though of course not everyone is the next J.K. Rowling.
This already differs from general beliefs about mathematics, where more people think that it is not ‘for
everyone’. Programming here is more aligned with mathematics. This belief also ties into construc-
tionist views on education, because if some people are born with the programming genes, they can teach
themselves in an exploratory fashion.

A closely related belief is the one that people that love programming, love the act and technology programming, rather than loving what programming can do for them. Again the comparison with language and mathematics help us place those beliefs. We learn to read and write for what it allows us to do, write love letters and petitions and dissertations and stories. Teachers stress this role of writing by having children read and write a variety of texts. With mathematics, of course it is also stressed what we can do with it, but, fewer people accept this, and the belied that there are people that love mathematics for mathematics is way more prominent than in language. Here too programming aligns with mathematics. This too aligns with an exploratory learning. If there are people that are born for programming and that love it naturally, we should not have a boring instructor stand in the way of their process!

This, of course, is a matter of ‘chicken and egg’. It is hard to know whether people believe this because they were implicitly taught about constructionism and thus believe or because the constructionist beliefs fit will into our community because of these prior beliefs.

2.4. Implications
Because of the constructionist tradition, the lack of a memory of programming lessons and the fact that people belief that some people are naturally drawn to programming leads to the fact that many programming lessons are what we call implicitly constructionist. Rather than being designed consciously constructed at exploratory teaching—which can surely have value and is regarded as a superior form of teaching by some, and as equally good as explicit direct instruction by others—people teach in a constructionist fashion by accident, without being aware of its strengths and weaknesses, and without considering alternatives. The role of this paper is not to reject constructionist teaching as a form of teaching, but it is to present direct instruction as precisely such an alternative. Similar to other fields where people argue about these core philosophies, this will happen in programming too, and that is fine. As such, we will present explicit direct instruction, summarize research that has found it beneficial and hypothesize how it could look like for programming.

3. Explicit direct instruction
Like with all teaching methods, are different forms of direct instruction and teachers follow methods with varying levels of strictness. The basic idea of direct instruction is that it is teacher-led, the teacher determines what happens and in what order, and directs information at students, hence the name directed. The lessons are structured and sequenced and have clear learning goals. One of the forms, which is often criticized is Direct Instruction with capitals (DI) as pioneers by Engelmann and Becker (Engelmann, Becker, Carnine, & Gersten, 1988). DI prescribes, among other things, quite rigid teacher scripts and that is probably where the bad name of direct instruction stems from. Scripts read aloud by robot-like teachers, that cannot be right! However more modern forms of direct instruction comprise a lot more than plenary lectures. Most notably Explicit Direct Instruction (EDI) as described by Hollingsworth and Ybara (Hollingsworth & Ybara, 2009) presents eight didactic principles that strengthen each other: They are clearly expressing the learning objectives, activating prior knowledge developing skills and developing concepts, explaining the importance of a lesson, and a sequence of guided practice with the teacher, practicing in groups at the plenary closure, followed by independent practice.
EDI lessons can be best summarized by Figure 1, where we see that an EDI lesson is more than the delivery of a canned script, it is a form of teaching that relies on metacognition of the teacher. They have to know what form of teaching they are using, why to use it and when to move to the next step (Hollingsworth & Ybarra, 2009). It is a form of teaching that activates students, and requires them to work together with the teacher in step 2 and together with peers in step 3. By no means does EDI mean that students sit back and listen.

3.1. Comparing direct instruction and exploratory learning

From its inception, proponents of direct instruction have presented its benefit in studies. For an overview of several early studies, we refer the interested reader to Binder and Watkins (Carl & L., n.d.). The debate still continues, but also more recent recent metastudies have demonstrated direct instruction to be more effective than less structured forms of teaching such as those based on exploratory, discovery or constructionist approaches (Klahr & Nigam, 2004; Kirschner, Sweller, & Clark, 2006).

What is more is that direct instruction can be an equalizing from of teaching. Irrespective of prior knowledge, all children are exposed to the same instruction, and practice together, so all students have the same opportunity to gain knowledge and to practice. In exploratory teaching, the students that already have some prior knowledge will be beter able to ask the right questions, and to attach new facts to existing knowledge, or to update mental models. Studies have demonstrates the performance of weaker students within a class especially suffers when exploratory methods are used, while they are the students that enjoy this form of teaching the most!

4. Explicit direct instruction in programming

Knowing that direct instruction seems to be a superior and more equal form of teaching, the final question that this paper aims to answer is ‘how would direct instruction for programming lessons look like?’ When we explore the seven design principles of EDI, how would we implement those in programming classes?

4.1. How programming differs from other subjects

So far we have (implicitly) assumed that programming and other fields such as language or mathematics are the same, and that teaching methods from those fields like direct instruction will also work for programming. While we absolutely believe the latter, there are some notable ways in which programming differs from other subjects which we will touch upon in the remainder of this section. For starters, when we present a student with a traditional programming environment, they can use all elements of the language. They can try to explore language concepts that have not been covered in class and try this. While of course, students in mathematics can also ask about percentages with you as a teacher are talking about multiplication, in programming students can venture on a path of their own. This inhibits, to a certain extent, the level of control teachers have over the learning trajectory of students. Secondly, compilers
and interpreters are relentless. When we are teaching grammar, we might disregard spelling errors for
the duration of the lessons to focus on grammar (not all teachers agree they should so unnoticed, but
most agree we can give errors outside of the scope of the current lesson or subject lower weight). When
we are teaching loops however, and a student has forgotten to indent their code consistently, Python will
not comply.

![Image](https://example.com/image.png)

*Figure 2 – The first screen students see when they try the Star Wars Hour of Code lesson.*

4.2. EDI design principles applied to programming
Taking the differences between programming and other subjects into consideration, we now inspect the
seven EDI design principles and explore how they are currently implements in programming lessons, or
how they could be.

4.2.1. Learning Objectives
EDI lessons start with a well-defined statement that explains what students will be able to be after the
lesson, in words that they can understand. So something like “after this lesson you will know monads”
is not useful when the students have not seen a monad before. Research demonstrates that the presence
of learning objectives increased performance (Hattie, 2008). Examining popular programming lessons,
we see that learning objective are missing in many cases. For example, consider the programming
environment of code.org, of which the first lesson is pictured in Figure 2. It is arguable a quite structured
and sequenced learning experience, where a learner is not exposed to all possible programming blocks,
but only a small subset. Even this learning environment though has contextual goals “getting the scrap
metal”, but does not explain the learning goal of the lesson.

4.2.2. Activating Prior Knowledge
By repeating existing knowledge in the beginning of a lesson, the teacher moves knowledge from the
students’ long-term memory to their working memory such that it can be used to connect new knowledge
too. Seeing the connection between different parts of programming, between concepts like a method and
a class, but also, at a lower level, the difference between and if and and if-else is important, and in this
respect we believe that programming education does relatively well. Concepts are presented together
and in relation to each other commonly.

4.2.3. Concept Development
Concept development means explaining the concepts that are present in the learning objective. So if the
objective is to learn monads, this part of the lesson concerns explaining what a monad is and when to use
one. This part is commonly present in programming lessons, in some cases, especially at the university
level, it is the only part used in the plenary part of teaching.
4.2.4. Skills Development

Skills development is teaching students the steps or processes that are used to teach the learning objective. It is an interesting aspect, because it raises the question what the skills are that one needs to master programming. One could argue that the most low level skills that children need are basic computer skills: using the mouse, clicking, using copy and paste and undo. In our experience teaching children programming, they rarely have those skills even in high school, and courses do not teach them. At higher levels, such as university, student might lack understanding of an operating system impeding their programming ability. A next step, closer to programming itself is the use of an IDE. That is surely a skill that helps students become more efficient in programming, and yet is not often addressed as topic of lessons or exercises. Looking at programming itself, there are skills (rather than concepts) that we also rarely teach, for example the correct use of syntax. In many programming lessons, as explained above, the concepts play a central role, while syntax is an afterthought. Students are assumed to pick up where colons go and when to use which brackets. Another example of a code skill in programming that we do not teach is reading and interpreting error messages.

At the highest level of skills, we could place strategies for programming. How is a problem approached and what steps does a professional programmer take when addressing a problem. We observe in programming lessons, that this too is something that is rarely vocalized or practiced.

We think skills development is the teaching practice where programming can benefit the most from direct instructional practices.

4.2.5. Lesson Importance

Lesson importance concerns explaining to students why the content of a lesson is important to them. While this might feel a bit boring, research shows that understanding the why of a lesson helps students engage (Hattie, 2008). Here too programming education could use some help. Where lesson importance is addressed, it is often within the scope of programming, for example “a loop is useful because it allows you to execute a computation multiple times”. This practice is probably related to the fact that programming believes that children have an interest in programming for the sake of programming, rather than in what it can help them create. Communicating why concepts and skills matter outside of programming (not an easy task!) will help students engage with programming lessons more.

4.2.6. Guided Practice

Guided practice is solving a problem together with students, represented by step 2 in Figure ????. In programming there is the practice of live coding, where a teacher or instructor programs in front of a classroom, but in many cases the plan for the live coding is made ahead of the lesson rather than collaboratively with the students. Also, problem solving strategies are not always discussed.

It is used in language education more commonly, where a teacher demonstrates, for example, how to write a story, while explaining their approach and asking students for input. A common technique for this is observational learning, where a teacher (or peers) demonstrates a task before learners attempt it. In writing education in fact, teacher modeling is the most prevailing way of using models for learning. Usually in the instructional phase (Koster & Bouwer, 2016). In this teaching method the teacher thinks out loud, they explain and demonstrate parts of the writing task. Pupils are expected to adopt the same line of reasoning when they will executing a writing task individually afterwards. It is shown an effective instructional method to teach writing strategies (see e.g. (Fidalgo, Torrance, Rijlaarsdam, van den Bergh, & Álvarez, 2015; Graham, Harris, & Mason, 2005; Koster & Bouwer, 2016)).

This effectiveness of this method is explained by the existence of the mirror neuron system in our brain. This system makes the brain demonstrate identical neural activity when we observe others performing a task as if we perform the task ourselves (see e.g. (Rizzolatti, 2005; Rizzolatti & Craigero, 2004)). In this way, the brain already ‘learns’ how to perform a task, and primes the execution of similar tasks.

This is definitely a didactic practice where programming could improve in a relatively easy way. From live coding it is not a large step towards making decisions and strategies explicit.
4.2.7. Lesson Closure
At the end of an EDI lesson, the teacher tests whether students master the demonstrated skills, of example with questions or by having them work on small, constraint problems. This has been explored for programming, for example in the form of worked examples (Gray, St. Clair, James, & Mead, 2007), but to say this is a practice that is common would be untrue. Most programming lessons both in schools and universities move towards independent practice soon after a plenary lecture.

It will not be all that easy however to start the practice of having students work on a small task in a collaborative fashion, since we are not used to teach in this way. Programming exercises, especially in the phase when students are still getting familiar with syntax, are most commonly performed alone. Also selecting or creating the right worked examples for the task at hand is not easy. While lots of improvement could be achieved by having students practice (and fail) in a more controlled environment, it will not be easy to convince teachers to do so. Here, the fact that students can use the entire programming language is a confounding factor. Maybe we need more programming systems where teachers can limit the possible statements and thus can exert more control, very much like code.org does it, but for text based languages, and controlled by the teacher.

4.2.8. Independent Practice
Independent practice is the part of a lesson where students work on larger programs independently. This part of learning is common in programming education, both at schools and at universities, but it is commonly the only part of practice that students encounter.

5. Concluding remarks
In this paper we have explored the constructionist history of programming, our lack of a shared memory of programming lessons, and related beliefs about programming, which we argue together have caused our field to teach in an exploratory way, without programmers being aware of this. We also summarize an alternative technique called explicit direct instruction, and present its benefits. We close the paper with a reflection on the current state of the art of programming through the lens of EDI and an outlook on how we could use more principles from direct instruction into our teaching.

6. References


well-taught lesson. SAGE Publications. Retrieved from https://books.google.nl/books?id=0iamQ21R8h0C


Phenotropic Programming?

Clayton Lewis  
University of Colorado Boulder  
430 UCB, Boulder CO 80309 USA  
clayton.lewis@colorado.edu

Abstract. In about 2002 Jaron Lanier began writing and speaking about phenotropic programming, in which programs aren't "simulations of vast tangles of telegraph wires" but instead rest on "the interaction of surfaces", using "pattern classification as the most fundamental binding principle, where the different modules of the computer are essentially looking at each other and recognizing states in each other, rather than adhering to codes in order to perfectly match up with each other." Lanier thereby hopes to combat "brittleness" of today's software, that makes software engineering "Sisyphean". These ambitions are consonant with ideas recently conspicuous at PPIG, as in Basman's (2016) paper, Building Software is Not a Craft. What has become of Lanier's vision in the last 15 years? This paper will review work that has been inspired by Lanier's ideas, and consider how further work may contribute to Lanier's goal of a "convenient [and] cognitively appropriate starting point for human beings who wish ... to go on to build things".

1. Introduction

In two related essays, “The complexity ceiling” (Lanier, 2002) and "Why gordian software has convinced me to believe in the reality of cats and apples” (Lanier, 2003), virtual reality pioneer Jared Lanier presented a critique of software, together with a diagnosis and a proposed remedy, called phenotropic software. The word "phenotropic" derives from the prefix “pheno-", referring to externals, or surfaces, and “tropic", meaning interaction. I'll label these two essays by their publication venues E (for Edge) and B (for Brockman) when quoting from them.

1.1 Critique

Lanier argues that software is unreliable, apart from small programs:

[I]t's been relatively easy in the history of computer science to make impressive little programs, but hard to make useful large programs. (E)

Accompanying the parade of quixotic overstatements of theoretical computer power has been a humiliating and unending sequence of disappointments in the performance of real information systems. Computers are the only industrial products that are expected to fail frequently and unpredictably during normal operation. (B)

He accuses computer scientists of idealizing software, creating a gap between what we expect of programs and what they actually deliver:

Wouldn't it be nicer to have a computer that's almost completely reliable almost all the time, as opposed to one that can be hypothetically perfectly accurate, in some hypothetical ideal world other than our own, but in reality is prone to sudden, unpredictable, and often catastrophic failure in actual use? (E)

1.2 Diagnosis

Lanier attributes the problems to choices made by the earliest thinkers in the field:
I've had a suspicion for a while that despite the astonishing success of the first generation of computer scientists like Shannon, Turing, von Neumann, and Wiener, somehow they didn't get a few important starting points quite right, and some things in the foundations of computer science are fundamentally askew. (E)

Their early thinking, he argues, was based on the transmission of signals over wires, an idea that does not provide an adequate basis for constructing computational systems as we need them to be:

If you go back to the original information theorists, everything was about wire communication. ... [This] might not have been the most convenient or cognitively appropriate starting point for human beings who wished to go on to build things. (E)

The limited idea of signals on wires has committed the field to the creation and implementation of protocols, conventions that define the form and meaning of sequences of signals:

At the dawn of computer science, in the mid—twentieth century, the only available intuition-building experience of information was the sending of pulses down wires. The early versions of information theory, which still dominate the standard curriculum, were concerned with single-point sampling of the world at the end of a wire. Therefore computer architecture as we know it was designed around simulated wires. Source code is a simulation of pulses that can be sent sequentially down a wire—as are passed variables, or messages. (B)

The way to make pulses on a single wire meaningful is to have a protocol that assigns meaning according to sequence. Most of the first half century of computer science was inspired by such protocols. (B) [emphasis added]

With temporal protocols, you can have only one of point of information that can be measured in a system at a time. You have to set up a temporal hierarchy in which the bit you measure at a particular time is meaningful based on "when" in a hierarchy of contexts you happen to occupy when you read the bit. You stretch information out in time and have past bits give context to future bits in order to create a coding scheme. (E)

Systems based on protocols have to devote resources to representing the protocols themselves, rather than the content with which the system should deal:

In order to keep track of a protocol you have to devote huge memory and computational resources to representing the protocol rather than the stuff of ultimate interest. This kind of memory use is populated by software artifacts called data-structures, such as stacks, caches, hash tables, links and so on. (E)

Further, the notion of protocol can only represent simple, limited interactions among the parts of a large information system:

Clearly protocol adherence is not an efficient means of explaining a system that receives a large number of inputs in parallel, and it is also probably an inadequate method of engineering very large systems. (B)

1.3 Remedy

Lanier proposes that protocols as a way of linking components of a system should be replaced by interactions at “surfaces”, structures containing many bits, that can be examined by other components:
The alternative [to protocols], in which you have a lot of measurements available at one time on a surface, is called pattern classification. In pattern classification a bit is given meaning at least in part by other bits measured at the same time. (E)

The components “connect to each other by recognizing and interpreting each other as patterns rather than as followers of a protocol that is vulnerable to catastrophic failures (E).” Pattern recognition, Lanier argues, places one in a regime of “approximation rather than perfection”:

With protocols you tend to be drawn into all-or-nothing high wire acts of perfect adherence in at least some aspects of your design. Pattern recognition, in contrast, assumes the constant minor presence of errors and doesn't mind them. My hypothesis is that this trade-off is what primarily leads to the quality I always like to call brittleness in existing computer software, which means that it breaks before it bends. (E)

Interaction by pattern recognition also opens up the possibility of “evolutionary self-improvement” in the interactions within a system. Each component would be trying to predict the states of other components, and receiving feedback on these predictions, enabling it to get better over time. Lanier argues that we should prefer the robustness of statistical interactions to the brittleness of idealized protocol interactions:

In the domain of multi-point surface sampling you have only a statistical predictability rather than an at least hypothetically perfect planability. I say “hypothetically”, because for some reason computer scientists often seem unable to think about real computers as we observe them, rather than the ideal computers we wish we could observe. Evolution has shown us that approximate systems (living things, particularly those with nervous systems) can be coupled to feedback loops that improve their accuracy and reliability. They can become very good indeed. (E)

1.3.1 Example

In (B) Lanier sketches an example of how phenotropic software might work. He envisions two teams of researchers, initially working independently, one of which develops a simulation of a lung, and one a simulation of a heart. The teams then decide to couple their simulations.

In the protocol regime of today, each team would define a protocol, likely an API, that the other team could use for interaction. The API would define what information the other team could access, and what control signals the other team could send.

Lanier describes two problems with this approach. First, he is skeptical that protocols could be defined that would cope with the complexity of the needed interactions between heart and lung. But, even if it were possible to define a workable protocol, “A working protocol would almost certainly reduce the prospects for improving any of the constituent organ simulations it connects.” The ideas about their simulations that the teams had at the time they defined the protocol could never be changed:

[S]oftware builds up in layers; it would be impossbly complicated and expensive to exhume protocols that have already been relied upon by many users in different ways. Thus we have the phenomenon of “lock-in,” in which some software becomes effectively mandatory. ... Beyond lock-in is an even more annoying software characteristic, which I’ve dubbed “sedimentation.” Software sedimentation is a process whereby not only protocols but the ideas imbedded in them become mandatory. ... As soon as the engineering groups have signed on to a protocol, the protocol becomes their master, since the groups would have to change simultaneously in order to revise it, and that would be effectively impossible because of the expense and complexity of the
task. Whatever ideas about interorgan communication are in vogue at the time of the protocol’s invention will be sedimented into place. Thinking will stop.

The phenotropic alternative is for each simulated organ to pretend... that the other was a real, physical organ being probed by real sensors. Each organ could measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and space. Each organ would appear to the other as a surface that could be sampled to varying degrees, but no higher-level parameters would pass between them. There would be no protocol other than the low-level one dictated by the nature of possible physical measurements.

For this scheme to work, each team would have to learn to recognize patterns in the other’s simulation. The heart would no longer be able to send a message that it had performed a beat. That would have to be inferred by the lung from such things as fluid motion and tissue displacements.

Lanier suggests that this paradigm could be extended to large software systems generally:

Perhaps in the future there will be an operating system whose components recognize, interpret, and even predict each other. Such a system would be less prone to catastrophic failure.

2. What has happened since 2003?

2.1 The immediate response

The essay (E) was accompanied by commentary, some of which was quite skeptical. The philosopher Daniel Dennett commented,

There are a few interesting ideas in his ramblings, but it's his job to clean them up and present them in some sort of proper marching order, not ours. Until he does this, there's nothing to reply to. (E)

Other critical commentary assumed Lanier to be talking about parallel computation, and answered what they saw as his arguments for that approach. Lanier responded that this criticism had “misfired”, because he was not talking about parallel computation; in fact (he points out) the word “parallel” does not appear in E.

The historian George Dyson suggested that Lanier was wrong to blame our problems on lack of vision among the founders of computer science (see also Note 1):

[It is unfair to attribute to Alan Turing, Norbert Wiener, or John von Neumann (& perhaps Claude Shannon) the limitations of unforgiving protocols and Gordian codes. These pioneers were deeply interested in probabilistic architectures and the development of techniques similar to what Lanier calls phenotropic codes. The fact that one particular computational subspecies became so successful is our problem (if it’s a problem) not theirs.

... The pioneers of digital computing did not see everything as digitally as some of their followers do today. "Besides," argued von Neumann in a long letter to Norbert Wiener, 29 November 1946 (discussing the human nervous system and a proposed program to attempt to emulate such a system one cell at a time), "the system is not even purely digital (i.e. neural): It is intimately connected to a very complex analog (i.e. humoral or hormonal) system, and almost every}
feedback loop goes through both sectors, if not through the 'outside’ world (i.e. the world outside the epidermis or within the digestive system) as well.” (E)

2.2 Later publications

Lanier has mentioned phenotropic software here and there since 2003, on his Web site (http://www.jaronlanier.com/), in some interview material in Rosenberg (2007), and, very briefly, in You Are Not a Gadget (Lanier, 2010). Google Scholar returned 104 hits for "phenotropic" on June 14, 2018, but most of these refer to unrelated uses of the term in pharmacology. Of those papers that refer to Lanier's idea, some contain only brief mentions or summaries of Lanier's work (Feiner et al., 2004; Ommeln, n.d.; Juba, 2011; Hissam et al., 2016).

A cluster of papers by Frénot, Hu, Privat, and colleagues applies the concept of phenotropic interaction to communication within a network of sensors and effectors (Hu, et al., 2011; Hu, et al., 2012; Privat, 2012; Hu, 2014). Such communication can be based on sensing physical quantities rather than on interpreting protocols, in some situations, as Lanier suggested. For example, an appliance like an oven can be identified by an energy management system by its pattern of energy consumption, with no participation of the oven in any protocol. The authors show how this approach can lead to greater flexibility in setting up and extending a network of devices. Some later work builds on these ideas (Jerde, 2017). Gabriel (2006) and Gabriel and Goldman (2006) include phenotropics as a mechanism for looser, more forgiving couplings between system components, as Lanier proposed.

Fleissner and Baniassad (2008) propose harmony-oriented architecture, inspired in part by the idea of phenotropic software. Systems in this framework are composed of snippets of code that communicate by diffusion of information within a spatial layout. Fleissner and Baniassad (2009 described a dialect of Smalltalk that works in this way. Martin (2011) describes a prototype implementation using Javascript. While this work goes some way to loosen the coupling between system components (some aspects of coupling can be changed by moving components without rewriting them) information exchanged between components has to be tagged, contrary to the phenotropic idea. The ideas of interaction by recognition, and the improvement of recognition and prediction over time, are not yet included.

3. Connections to ideas of Basman and colleagues.

In a series of papers at PPIG and elsewhere, Antranig Basman and colleagues have presented ideas that align in some ways with those of Lanier, as critique and diagnosis (Basman et al. 2015; Basman, 2016; Basman et al., 2016; Basman, 2017; Clark and Basman., 2017; Basman et al., 2018).

3.1 Critique

In “Building Software is Not [Yet] a Craft”, Basman (2016) notes that in the “Computer Science of the Present, a product may spontaneously disintegrate without warning, suddenly becoming wholly unusable.” Compare Lanier (as quoted earlier) on software being “prone to sudden, unpredictable, and often catastrophic failure in actual use (E).” In the same essay, Basman also cites the “the potential for the complexity of artefacts to exceed our ability to manage or comprehend them;” compare Lanier (B), “Since the complexity of software is currently limited by the ability of human engineers to explicitly analyze and manage it, we can be said to have already reached the complexity ceiling of software as we know it.”

In "If What We Made Were Real", Basman (2017) calls for "software which is real, in that it behaves with the same continuity and consistency as real materials — real trees and real mountains expose a consistent and coherent set of linked aspects, affordances and appearances as we move from place to place, scale to scale and sense to sense." Compare Lanier's description (quoted earlier) of the heart and lung simulations that relate as if each "was a real, physical organ being probed by real sensors. Each
organ could measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and space. (B)"

3.2 Diagnosis

In one respect Basman’s and Lanier’s diagnoses are virtually identical, though arrived at independently. Lanier (E):

The reason we’re stuck on temporal protocols is probably that information systems do meet our expectations when they are small. They only start to degrade as they grow. So everyone’s learning experience is with protocol-centric information systems that function properly and meet their design ideals. This was especially true of the second generation of computer scientists, who for the first time could start to write more pithy programs, even though those programs were still small enough not to cause trouble. Ivan Sutherland, the father of computer graphics, wrote a program in the mid 1960s called "Sketchpad" all by himself as a student. In it he demonstrated the first graphics, continuous interactivity, visual programming, and on and on. Most computer scientists regard Sketchpad as the most influential program ever written. Every sensitive younger computer scientist mourns the passing of the days when such a thing was possible. By the 1970s, Seymour Papert had even small children creating little programs with graphical outputs in his computer language "LOGO". The operative word is "little." The moment programs grow beyond smallness, their brittleness becomes the most prominent feature, and software engineering becomes Sisyphean.

Compare Basman (2017):

We got into this mess through 60 years of consistently drawing the wrong people into our field, and continuing to entrench its vices rather than reform them. In the “Garden of Eden” phase of Computer Science when such inspired products as McCarthy’s Lisp and Sutherland’s Sketchpad were plentiful as tabby cats, it was easy to imagine that maturing to solve more ambitious problems for a wider class of people was just a step away. In a world that has given us Java, Haskell and Ruby, success seems further away than it ever has been.

3.3 Remedy

As we’ve seen, the key elements of Lanier’s remedy are interaction based on surface interactions, rather than protocols, and self-improving recognition as the basis of communication. Basman and colleagues emphasize different themes, especially support for open, ongoing authoring of software systems (Basman et al., 2015; Basman et al., 2016). But there are nevertheless some points of agreement.

Lanier’s idea of system components “pretending” to be physical objects, and interacting by measurement rather than by protocol, is related to Clark and Basman’s ( emphasis on externalized state transfer between system components. Traditional software components ordinarily conceal as much of their state as possible; a normal role for APIs is to expose as little state as possible. Clark and Basman argue that components should instead expose as much state as possible. Physical objects, or software components “pretending” to be physical objects, do the same. Lanier’s cautionary story about the fate of the heart-lung simulation, “locked in” and “sedimented” once protocols are in place, can also motivate Clark and Basman’s (2017) open approach.

In "An Anatomy of Interaction", Basman et al. (2018) present a vision of a system in which system state mirrors the state of some portion of “the world of interest”. They go on:
opening up the design process requires this mirroring to have an open structure. Each community of interest will have different concerns that lead them to select different sources of state, and mirror them with different representations. We require that both the description of the state to be materialised, as well as the materialised state itself, are externalised [in a way] that is available throughout the lifetime of the system, not just an early design phase.

This is a description of system components that are “pretending” to be physical objects, in Lanier’s terms, and carrying on pretending, and being open to interaction as such.

Physical objects can not only be observed but also be modified. In "If What We Made Were Real", Basman (2017) contrasts the way this can be done, in physical craft activities, with how today’s software behaves when worked on:

Scratch the surface of a physical product such as a chair or a wall, and you find something broadly similar underneath. The physical world is worked on by means of tools that are part of its own idiom — whether we cut a piece of wood into a smaller piece of wood, or make a hole to hold a bracket, we are using the affordances of the world itself to cause change. Contrast this with the nature of a modern piece of software or hardware — scratch the surface and underneath it is an incomprehensible world of blinking lights and mass of wiring that bears no resemblance to the physical form and affordances of the overall object (see Note 2).

Further, Basman (2016) describes how physical materials respond to change in a continuous, rather than a discrete way:

Each state of the material is closely surrounded by a dense collection of neighbouring states that behave similarly. ... However, today's software materials, traditionally consisting of source code text in an editor buffer, could not be more sparse. The “nearest neighbouring program” to any given one is separated from it by a vast ocean of syntactically invalid or crashing variants (see Note 3).

As quoted earlier, the virtues of “approximation” for Lanier require a related continuity, though not expressed explicitly:

When you de-emphasize protocols and pay attention to patterns on surfaces, you enter into a world of approximation rather than perfection. ... Pattern recognition ... assumes the constant minor presence of errors and doesn't mind them. (E)

The link to continuity is that “approximation” is only valuable if things that are similar in perception are similar in meaning, so that small errors don’t matter.

While we can see some resemblances between Lanier’s conceptions and those of Basman and his collaborators, there are differences in approach. As mentioned earlier, Basman and collaborators focus their efforts on supporting more flexible authorship of software, by people. By reducing the barriers to understanding and modifying software (often deliberately erected) they hope to create an ecosystem of software that develops more organically, and fails less catastrophically. Problems can be repaired by moving a little way to a nearby state, without requiring wholesale reworking.

By contrast, Lanier hopes that recognition-based interfaces between system components will improve themselves, as each component seeks to develop a better and better model of the behavior of its neighbors. There’s no explicit role for human authors in this picture. Lanier hopes that software engineers working with components coupled in the proposed more robust way will find that the systems they build will turn out to be more robust.
One way to probe the prospects for this hope is to explore the idea of “pretending” to be a physical object. If Lanier’s heart simulator actually were a heart, it would exhibit the continuous response to intervention that physical objects have. Looking behind its surface, the structure that it presents to the lung simulor, one would find real physical substance, as Basman notes. But if the simulator is only “pretending” to be a heart, that is, only displaying the right behavior on its surface, what then?

One possibility is that one would find the “incomprehensible world of blinking lights and mass of wiring” of today’s software behind the surface. In that case the support for human tinkering within the “pretending” heart simulator would be no different from what is available now. The benefits of the approach would come from looser coupling outside the simulator, together with the prospect of automatic improvement in this coupling.

Alternatively, one might find other surfaces behind the outer surface of the simulator. Lanier says little about this. He does propose that there could be “an operating system whose components recognize, interpret, and even predict each other, (B).” This suggests that smaller, as well as larger, software components could be phenotropic. We return to this possibility below.

Another aspect of Lanier’s proposal is that phenotropic interfaces are meant to improve themselves, without the human authorship with which Basman and colleagues are concerned, by a process of feedback-guided evolution. Lanier sketches how this could work in E, using the example of a system that creates a graphical model of a person, an avatar, from video, recovering the geometry of the person, and how they are moving in space, relative to the camera. There’s also a system that uses this model to produce video that shows the avatar moving. If both systems are successful, the video of the avatar should closely resemble the original video. Lanier proposes that the system that builds the avatar can improve its processing of the input video by using feedback about the video reconstructed from the avatar.

The avatar example contains an encoder-decoder pair: the system that creates an avatar from video, and the system that creates a video from the avatar. Such pairs can indeed tune themselves, as is common in machine learning work (see e.g. Cho et al, 2014). How would this work in other situations?

Lanier applies the idea to the heart-lung simulation:

Each team would also learn to build a model of the other organ, in order to assist in interpreting measurements. These models might not exist as independent, separable structures but might be implicit in the chosen signal-processing methods, and would almost certainly be able to retune themselves with use. (B) [emphasis added]

Here there is no preexisting decoder, corresponding to the system that produces video from avatars in the avatar example. Lanier may be suggesting that the connection between the heart and lung simulations is an autoencoder (https://en.wikipedia.org/wiki/Autoencoder); these can indeed tune themselves. The autoencoder would map the information available on the surface of the heart to a lower dimensional representation, with the requirement that the map can be inverted with small loss. By including temporal information the autoencoder might recover a dynamic model of the heart adequate to predict the timing of changes on the heart’s surface, that might correspond to beats. Thus, as pointed out earlier, no protocol would be needed to communicate beats from heart model to lung model.

It’s not clear what work this does, though. The lung model does not need to know about beats, as such. It needs to know the pressure produced by the heart as it pumps, and has to relate this pressure to the pressures induced in the lung. But Lanier already requires that each model can “measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and
space”, as said earlier. Given that these measurements are available, the need for recognition, and thus
the value of retuning, isn’t clear.

4. The psychology of phenotropic programming

Lanier’s root concern is that when two systems try to communicate they should succeed despite failures
of detail. The appeal of “recognition” is that it succeeds in the presence of noise. The appeal of retuning is
that the ability of recognition to reject noise can improve over time. The appeal of “measurement” is that it
suggests communication that is tolerant of errors; quantities can be measured approximately without
losing all meaning.

If we take the idea of “surface” to be simply a structure to which it makes sense to apply recognition,
which seems broadly consistent with Lanier’s thinking, we can see examples of phenotropic
communication in systems today, especially at the user interface. Consider looking up a book in a library
catalog. I’ve just verified that I can search for books by “jaron lanier”, “lanier, jaron”, “lanier jaron”, “Jaron
Lanier”, or “Lanier, Jaron” in my library catalog, and get the same results. If I ask for books by “lanier
jared”, the catalog asks if I meant “lanier jaron”. This is a concrete instance of communication succeeding
in the presence of noise. The catalog is not simply searching for what I specified, it is recognizing what I
may have meant. What I typed is the “surface” of me that the catalog responds to.

Amazon’s surface for my book searches is considerably bigger. If I log in, as I’ve done just now, it finds
books for me that I didn’t search for at all, based on aspects of my history of which it is aware. One of the
suggestions is actually a book a friend just recommended for me; I have no idea what was on my surface
that enabled that.

Retuning is constantly at work in Amazon’s recognition system. This isn’t autoencoder tuning, but a kind
of reinforcement learning: Amazon’s recommendations, that is, its recognition of my interests, are
reinforced when I act on one of them. The recognition system is tuned by my responses, and those of
jillions of other consumers.

Can these ideas get any traction within the realm of programming? A few crumbs of this kind of behavior
have long been present. The ability to write 1 + 2.5 rather than 1. + 2.5 was not supported in FORTRAN
at first. If you intended 1 to be interpreted as a floating point quantity you needed to write a floating point
literal, with a decimal point. Happily nearly all compilers now “recognize” that someone who writes 1 + 2.5
wants floating point arithmetic, and insert the needed conversion.

But the surface involved here is miniscule, and corresponding “coercions”, or automatically inserted type
conversions, are quite rare in other situations. Nearly always a programmer has to do the work of
modifying an intention to fit the precise form of an allowed request. This is the kind of protocol fiddling that
Lanier wants to eliminate.

For example, suppose one has a function, foo, that takes two coordinates, x and y, as arguments. If one
has a point, p, given as a pair of coordinates, in most languages one can’t simply provide the point as an
argument to foo. Rather, one has to write something like foo(p[0],p[1]). If one’s interaction with foo was
conducted via a surface, foo might be able to recognize what to do, if just p appears on the surface.

Could foo’s ability to respond as desired to a range of surface contents improve over time? It doesn’t
seem that the autoencoder idea, as an unpacking of Lanier’s intention, can work here. But reinforcement
learning, of the kind Amazon uses, could be applied. If foo’s conjectured recognition can be approved or
rejected by the programmer, its ability can be tuned. Such tuning could take place for a whole community
of programmers, just as Amazon’s tuning uses results from many consumers, or it could be individualized.
For example, foo might learn something reasonable to do if given just a single number (duplicate it? pair it with a zero?).

The surface foo examines could be broadened beyond the arguments specified for it, to include the surrounding code. In the example above, when foo is given a single number, the best response could depend on what else is happening in the code.

This small example stands for a much larger, more consequential population. In many programs the number of characters that correspond to the programmer’s overall intentions, those that relate to entities in the outside world, is quite small, relative to the size of the program. That is, a great deal of work goes into internal considerations, including many variations on the theme of this example, which is putting things into structures and taking them out again. These are the “stacks, caches, hash tables, links and so on” whose necessity Lanier attributes to protocol-centric computing. What might be called the payload ratio problem is that too much software doesn’t express the purpose of code, but just the implementation of it in a specific setting (see Note 4.)

Would software made up of components like foo have the merits of physicality, that Lanier and Basman and his colleagues call for? The answer may depend on the stability of the resulting recognition behaviors. What’s wanted is a regime in which similar programs do similar things, yes, but also what they do is predictable. Part of the appeal of measurement in Lanier’s thinking is that the meaning of a gram or a second, unlike the meaning of a pointer, does not change over time. Software structures anchored in these stable ideas can themselves be stable. Can phenotropic software structures that tune their behavior over time be stable, too?

The perspective of the payload ratio problem suggests that they might. Because programmers’ intentions are often grounded in entities in the world, a facility that squeezes out other stuff from the code, as phenotropic structures may do, may leave a residue that is more stable, and easier to understand. This could be true despite the fact that the implementation of these structures, relying as it would on complex and even dynamic recognition behavior, would be very complicated, and indeed opaque. But this situation is actually quite natural, and familiar; it’s what we experience with our fellow humans, and we are famously more robust than the software systems of today.

**Notes**

Note 1. In his commentary on E, Dyson suggests that the brittleness and unreliability of software, of which Lanier complains, may actually be virtues:

> I'm not immersed in the world of modern software to the same extent as Jaron Lanier, so it may just be innocence that leads me to take a more optimistic view. If multi-megabyte codes always worked reliably, then I'd be worried that software evolution might stagnate and grind to a halt. Because they so often don't work (and fail, for practical purposes, unpredictably, and in the absence of hardware faults) I'm encouraged in my conviction that real evolution (not just within individual codes, but much more importantly, at the surfaces and interfaces between them) will continue to move ahead. (E)

This perspective builds on ideas Dyson presents in his book, Darwin among the Machines (1997), about the evolution of computational structures. While these structures are not self reproducing, and not free living, they replicate with enormous fecundity within an environment, increasingly shaped by them, that is increasingly hospitable to them. Where once code could execute in only a handful of processors, and could be replicated only with considerable effort, much less than a century from its first appearance, code can travel around the world at the speed of light, and execute in millions of hosts. Just as random
variability fuels biological evolution, so brittleness and bugs may be important to the longer-term development and spread of computational structures:

No one can say what contribution randomness has made to software development so far. Most programs have grown so complex and convoluted that no human being knows where all the code came from or even what some of it actually does. Programmers long ago gave up hope of being able to predict in advance whether a given body of code will work as planned. ... The software industry has kept track of harmful bugs since the beginning -- but there is no way to keep track of the accidents and coincidences that have accumulated slight improvements along the way. -- Dyson (1997), p. 124.

Note 2. In E Lanier describes today’s software as “simulations of vast tangles of telegraph wires”.

Note 3. Attaining the desired denseness, or continuity, in a space of software necessarily requires giving up much of the “power” of traditional computation, that is, the “power” to have arbitrarily small changes cause arbitrarily big effects. In "If what we made were real" (2017) Basman lists other powers that should be sacrificed, including “The power to construct programs that might consume unbounded time and/or space, or perhaps never terminate,” and “The power to prescribe the exact sequence of operations needed to achieve a particular result.”

Note 4. Bret Victor deals with these matters in his brilliant (and poignant) historical pastiche, The Future of Programming (2013), speaking as if in 1973:

So, say you’ve got this network of computers, and you’ve got some program out here that was written by somebody at some time in some language; it speaks some protocol. You’ve got another program over here written by somebody else some other time, speaks a totally different language, written in a totally different language. These two programs know nothing about each other. But at some point, this program will figure out that there’s a service it needs from that program, that they have to talk to each other. So you’ve got these two programs—don’t know anything about each other—written in totally different times, and now they need to be able to communicate. So how are they going to do that? Well, there’s only one real answer to that that scales, that’s actually going to work, which is they have to figure out how to talk to each other. Right? They need to negotiate with each other. They have to probe each other. They have to dynamically figure out a common language so they can exchange information and fulfill the goals that the human programmer gave to them. So that’s why this goal-directed stuff is going to be so important when we have this internet—is because you can’t write a procedure because we won’t know the procedures for talking to these remote programs. These programs themselves have to figure out procedures for talking to each other and fulfill higher-level goals. So if we have this worldwide network, I think that this is the only model that’s going to scale. What won’t work, what would be a total disaster, is—I’m going to make up a term here, API [Application Programming Interface]—this notion that you have a human programmer that writes against a fixed interface that’s exposed by some remote program. First of all, this requires the programs to already know about each other, right? And when you’re writing this program in this one’s language, now they’re tied together so the first program can’t go out and hunt and find other programs that implement the same service. They’re tied together. If this one’s language changes, it breaks this one. It’s really brutal, it doesn’t scale. And, worst of all, you have—it’s basically the machine code problem. You have a human doing low-level details that should be taken care of by the machine. So I’m pretty confident this is never going to happen. We’re not going to have API’s in the future. What we are going to have are programs that know how to figure out how to talk to each other, and that’s going to require programming in goals.
References


Basman, A. (2017) If What We Made Were Real: Against Imperialism and Cartesianism in Computer Science, and for a discipline that creates real artifacts for real communities, following the faculties of real cognition. Proceedings of the Psychology of Programming Interest Group.


Investigating Conversational Programming for End-Users in Smart Environments through Wizard of Oz Interactions

Kate Howland  
Department of Informatics  
University of Sussex Brighton, BN1 9QJ, UK

James Jackson  
Department of Informatics  
University of Sussex Brighton, BN1 9QJ, UK

Abstract
Natural language programming has long been an aspiration, but is fraught with challenges that have so far prevented genuinely useful and useable applications. End-user programming for smart environments is increasingly being pursued through trigger-action rule services that include simplified natural language description of rules. Along with the increasing prevalence of Voice User Interfaces (VUIs) in smart environments, this points to new opportunities for supporting understanding, debugging and editing of rules through speech. However, there is a lack of contextually relevant data on how end-users without programming experience describe and understand rules for smart environment behaviours through speech. This paper describes how the CONVER-SE project is developing methodology and software for capturing this data, and prototyping VUIs that attempt to mitigate the many challenges with supporting programming interactions through speech in this context.

1. Introduction
Programming using natural language long been a goal in end-user and novice programming research, but has so far fallen short of expectations due to fundamental challenges in reaching alignment in communication between human and system. VUIs such as Amazon Echo/ Alexa and Google Home/ Assistant have made speech a frontrunner for smart home control, but do not yet support editing, debugging and authoring of smart home automation rules through speech. Understanding, configuring and customising the rules that define smart environment behaviours are end-user programming (EUP) activities. Currently, these activities must be done using a separate, screen-based interface, as voice interaction is largely limited to triggering pre-defined behaviours. Automation platforms such as IFTTT and Stringify allow programming of smart home behaviours through trigger-action rules, but have seen limited uptake beyond early adopters and tech-savvy hobbyists. These platforms provide natural language descriptions of the trigger-action rules, but the rules are detached from the context in which they will be carried out. There is a gulf between abstract representations of automated behaviours and the concrete real-world environments in which they play out. For example, a user standing next to a smart lamp wanting to understand or reconfigure the rules for its behaviour must turn their attention from the room to a screen, understand and edit a code-like description, and draw a link between a unique identifier and the object in the room. Supporting these activities through a voice interface, with potential to include gesture and proximity data to support disambiguation, could provide more intuitive ways of understanding and programming smart environments.

With VUIs now widely used in intelligent assistants, there is renewed interest in programming through speech, but we lack foundational research on how users without a programming background can best understand and express rules defining smart environment behaviour.
Gathering data on how end-users naturally express programmatic rules is a well-established approach in EUP research. However, studies of natural expression of programmatic rules for smart environments are typically carried out using toy scenarios in decontextualised settings, and often limited to written responses to survey questions or similar text-based descriptions. This means that there is very little data on natural expression of rules through speech, and no data on how co-speech gesture and contextual elements such as proximity support speech when describing rules. In smart home scenarios, the presence of cameras in sensor-enabled environments makes it feasible for additional contextual information to be used to resolve ambiguities and deictic references (e.g. this, there, that). In addition, it is important to recognize the extent to which ‘natural’ expression is increasingly influenced by expectations from interaction with existing similar systems. In the context of conversational interfaces, it may be more realistic to focus on language alignment between the system and the user.

In the CONVER-SE project, we are examining the challenges of speech programming for smart environments, and investigating how these could be mitigated in a conversational interface. To carry out this research, we are developing methodology by adapting natural expression studies to include capture of speech, gesture and proximity in situ. We are making use of Wizard of Oz prototyping (in which some or all functionality is implemented by a human) and participatory methods such as bodystorming (in which participants play out interactions with an imagined future system). This paper describes the design of a domestic study that is currently in progress, and the development of a software toolkit for Wizard of Oz prototyping of conversational interfaces.

2. Background

Previous research on EUP for smart environments has gathered natural language descriptions of rules using empirical methods including online surveys [1, 2], post-it note instruction tasks [3] and interviews [4]. Existing work has led to some consensus, including trigger-action rules as a simple but powerful format [2, 5], an inclination for users to rely on implicit rather than explicit specification [1, 2] and a tendency for them not to mention specific sensors or devices [1, 2, 4].

Although these studies have provided important insights into the natural expression of tasks and rules for smart environments, context has been largely overlooked in this work, and none of the studies were conducted in real-world scenarios. In addition, natural language descriptions have been collected in isolation from other communicative modes, such as gesture. Given the importance of context for smart environments, it is likely that existing findings only provide a limited picture. For example, the finding that end-users do not make reference to specific sensors or equipment, first reported by Truong et al. [1] and validated by the findings of Dey et al. [4] and Ur et al. [2], may well have been influenced by the lack of real-world context in the studies. Referring to sensors that you know exist in your house would be much more likely than referencing hypothetical sensors in a toy scenario. The importance of real world contexts for smart environment EUP research is beginning to be recognized. For example, a recently published EUP study comparing different notation styles for home automation was carried out in real domestic environments [6], but unfortunately the study design did not allow for examination of contextual referencing, or capture of speech, gesture or proximity data. Our own lab-based pilot work [7] also suggested a number of limitations with decontextualized studies, as described further in the following section.

3. Domestic Studies

To gather contextually valid data, we are conducting our studies in participants’ homes. We are recruiting study participants who already have some level of smart home technology, but who do not consider themselves technology experts, and have no previous programming experience. We are particularly keen to reach ‘inadvertent adopters’, who we define as the family and housemates of early adopters.

Our first study, which is currently underway, is designed to investigate the following questions:

- How do end-users naturally understand and specify rules for smart environment behaviours in their own homes?
- What ambiguities and inaccuracies are present in understandings of and expressions of such rules?
To what extent does participants’ language align to that used by a VUI over the course of an interaction?

How far can conversational approaches help with understanding, editing and generating complete and unambiguous rules?

We are investigating these questions through a three-part study, which is video recorded. Part 1 is a semi-structured interview that investigates participants’ current use and understanding of smart home technology and VUIs, and captures natural descriptions of rules for automated behaviours they would like to have running in their homes. Part 2 involves Wizard of Oz prototype interactions, allowing us to test conversational approaches to supporting editing and generating of rules (including modelling of rule structure and stepwise composition of rule parts). This also allows us to examining if/how conversational alignment occurs over the course of interaction with the VUI. Part 3 involves participants bodystorming future interactions with a more advanced VUI that could detect contextual information, allowing us to seek active input from participants on ideas for effective support.

3. Wizard of Oz Toolkit

For our studies we needed an application that allows us to simulate the behaviour of an advanced VUI designed to facilitate EUP of smart devices, in order to test the effectiveness of different conversational approaches. For the first study we use a Bluetooth speaker and light, controller remotely through a laptop, as shown in Figure 2. As we are also interested in assessing the benefits of providing visual feedback as part of the EUP process in future studies, we included a requirement that the application be capable of projecting text and images to a mobile display device. The final requirement was that the core application functionality be available offline, as the studies are to be carried out in situ, where the availability of an internet connection cannot be assumed. For speech production, we reviewed several offline text to speech (TTS) engines that could be integrated with the application but were dissatisfied with the quality of the generated speech. We therefore decided, at least for the initial version of the application, to use an online TTS service to create a database of audio clips that could be linked together in different ways to generate the utterances we anticipated might be required: requesting information, making statements, checking understanding and confirming when actions are completed. The utterances were structured so as to minimise the required number of audio clips, while providing a flexibility of response and not unduly compromising the production of natural sounding speech. The majority of the audio clips are required for time and event triggers e.g. “At 7:30pm every day” or “when someone enters the room”; a smaller number cover general statements and questions e.g. “I’m sorry, I don’t understand.” or “What would you like to change?” and finally there is a group of clips specific to each device e.g. “turn on the kitchen radio and select Radio 3” or “switch off the bedroom light”.

The application was designed to automate functions where possible so the researcher’s attention could be focused on selecting the most appropriate VUI response. Therefore depending on the selection function (e.g. “check” or “confirm”), the application will automatically generate the required composite audio clip from the database of stored clips and display the audio text for review before the researcher plays the clip. Changes to the ruleset are automatically tracked and there is selectable option to check for rule conflicts and duplicate rules. If required the ruleset can be projected to a remote display. After testing an early prototype of the application, a filters panel was added to the dashboard to enable users to limit the displayed triggers and actions (Figure 3).
To help with data collection and analysis, the application can make an audio recording of the study session and an event log records and timestamps all application activity.

Figure 3: Dialogue Manager showing Dashboard panel

4. Conclusions and Further Work
The first study is ongoing, with video data from the first participants currently being transcribed. From our observations so far, we note that participants find the stepwise composition of rules with conversational prompts from the interface relatively easy, whilst the statement of complete rules without the assistance of prompts is highly demanding, in line with expectations from analysis of speech-based interaction according to the Cognitive Dimensions of Notations [3. p.5]. Once we have finished data collection, we will begin analysis to answer the research questions stated in section 3, and use the findings to drive the design of conversational support for our next prototype.

5. References
Crafting Design Documents in First-Year CS Courses

Shannon Ernst
Oregon State University
ernstsh@oregonstate.edu

Jennifer Parham-Mocello
Oregon State University
parhammj@oregonstate.edu

Abstract
Computer science, at its core, is about solving problems. The "Carry out the Plan" portion of problem solving is often examined and emphasized in CS 1 and CS 2, forgetting to emphasize the other important aspects of the problem-solving process. This study focuses on the "Devise a Plan" or design step of problem solving. Four terms of design data (2,797 designs) are examined to answer the question of whether syllabus detail impacts the crafting of design documents and what are the students' attitudes toward design. The results show that syllabus detail does impact the way students design and that students do value design when asked in a survey. These insights have implications of when and how design is taught and opens questions to how best assess design.

1. Introduction
Computer science, at its core, is about solving problems. As educators in computer science, the general goal should be to teach students a systematic approach for solving problems (Raadt, 2004). Often the focus of solving the problem, especially in the initial stages of computer science, is on programming in a particular language. In early, first-year computer science classes, the emphasis is placed on how to structure the code and how to write syntax, and this emphasis largely misses the bulk of the problem-solving processes. As defined by George Polya in How to Solve It (1957), there are four steps used to solve problems: 1) Understand the Problem, 2) Devise a Plan, 3) Carry out the Plan, and 4) Look Back. In many freshman computer science courses, the step to “Carry Out the Plan” typically involves writing the code/program for an assignment, and the grade for the assignment is based on this step, which removes focus from the other three steps in the problem-solving process. However, one recent study used Polya’s problem-solving steps to teach software development and found that students exposed to these steps were more capable to solve problems and did so in a shorter amount of time (Allison and Joo, 2014). Grounded in Polya’s theoretical framework for solving problems, the action research in this paper uses student engagement and attitudes in the first-year courses to evolve a strategy and rubric for teaching the craft of creating design documents prior to coding and before learning formal software engineering practices.

2. Related Work
As Ginat et. al. point out, there is an abundant amount of literature on how students understand programming and computer models, but there are few studies on how students perform task analysis and utilize patterns (2013). The Ginat et. al. (2013) study looks at students’ utilization of patterns; whereas, the study presented in this paper focuses on students’ ability to recognize and perform tasks presented in a design document template and grading rubric. Instead, we are asking them to recognize the goals of the problem by listing the requirements and assumptions, as well as “put it all together” in a design plan, prior to coding and thinking about syntax. According to Soloway (1986), we are asking for the explanation using stepwise refinement, but we are also asking for students to combine these goals/plans with rules without using a specific programming language.

“To facilitate the transfer of knowledge from “Computer Science 100” to other problem-solving activities, students must be taught explicitly that programming is a design discipline, and as such the output of the programming process is not a program per se, but rather an artifact that performs some desired function.”

We are interested in Soloway’s plan composition, but unlike the Fisler et. al. (2016) study, we are interested in the crafting of the plan composition prior to coding, rather than identifying the plan composition presented in code. At this time, we do not investigate students’ plan composition used in the design documents because the focus here is on establishing a curriculum that promotes planning and design early in the first-year CS classes.

Other related work tends to revolve around a multi-institutional study published in 2004 where students were asked to design a super alarm clock (Chen et. al., 2005), (Eckerdal, 2006), (Fincher,
The study was conducted at 21 institutions, across four countries and garnered 314 participants with half the students being early in their computer science program and the other half preparing to graduate. The designs were collected in a controlled and timed setting, and the results from these studies show that design is a skill that is acquired and improved over time. Though these studies have a wider breadth in regard to the institutions and backgrounds of the students, the research presented in this paper does not collect data in a controlled, timed environment because the authors’ want to investigate the craft of creating design documents in one’s own natural environment. In addition, the authors in this research study seek to teach problem-solving techniques and skills to first-year computer science students, rather than allowing them to acquire these skills. The common thread of the multi-institutional studies is they assume the design of a solution is constrained to UML diagrams or formalized notation. Whereas, this study investigates unofficial notation wherein students and professionals express their thinking in a variety of ways which can still address the key points of design and yield viable software solutions. Our research aligns a little more closely with a recent study analyzing novice student design strategies using recorded sessions of students creating design documents (Yeh, 2018); whereas, we are more interested in what students do on their own outside of a recorded environment over the course of multiple assignments.

Work published in the programming language community focuses on design recipes. These design recipes come from How to Design Programs and is referred to as Program by Design (Felleisen, 2001). Recent research used the Program by Design method in an introductory class and focused heavily on the link between design and code with the belief that there are patterns students should employ each time they design a function (Sperber and Crestani, 2012), (Ramsey, 2014). The work examined in the presented study does not follow a set step by step design process nor does it examine the links to code in as much depth as work found in relation to Program by Design. Rather, the students in the presented study are encouraged to engage in whatever form of representation best lets them learn, whether that is pictures, text or some combination. The idea behind this freedom of choice is to not bog students down with learning formal notation for design so early in the process to the point where they may not see or examine different ways of solving the problem.

There is also work on test driven design as an alternative to recipes and other systematic approaches (Janzen and Saiedian, 2008), (Proulx, 2009). It was found that students who engage in test driven design see more benefits when programming however many students are reluctant to adopt test driven design, instead preferring the test last approach (Janzen and Saiedian, 2008). In the presented study, a testing table is submitted as part of the design before the programming assignment is submitted, attempting to encourage students to reflect before and after implementation. The design in this study is not marketed under a test first policy though.

The classification system used in this study was based largely on a combination of Polya’s steps and the rubric established by Thomas et al. (2014). Other rubrics to assess design quality exist, such as Castro and Fisler’s SOLO Taxonomy (2017). Their rubric, though very detailed, was more closely coupled with how the code was structured and interacted rather than the designs represented in this paper which do not rely on examining the student’s code structure (Castro and Fisler, 2017). Their study was also very small with only 15 participants. The presented study continues to examine the correlation between designs and grades and the inclusion of design attributes, outlined by the authors, but over a much larger data set. There are many more studies that focus on the detection of student patterns used in a solution (Ginat, 2009), (Ginat and Menashe, 2015), and use the SOLO taxonomy to classify the quality of the patterns or building blocks used by the students in an algorithmic design or actual code written in a programming language (Ginat and Menashe, 2015), (Izu et. al., 2016). However, this research does not focus on detecting patterns or classifying the learning based on these patterns. However, this research study is interested in determining a way to get first-year students to engage in multiple problem-solving steps prior to writing code, instead of skipping steps or engaging in them after code writing occurs.

3. Motivation
Systematic problem-solving prior to coding has been the primary motivation for this research, and before this study was conducted, first-year students at the host university had been required to submit
designs with their programming assignments for the past 4 years. These designs were not meant to be formal UML write ups and were not expected to be correct. The purpose of the design was to encourage students to think about the problem before they began programming the solution. At that time, students had access to a description of George Polya’s problem-solving steps (see Table 1) and an example design document for what to include given a specific problem statement (see Figure 1). The example design document provided both a flowchart and pseudocode as a means for devising a plan, but the students were not required to do both. However, the students were shown that the example design document does not include code, and they were provided many more test cases to show good test coverage for good and bad test cases.

The students were reminded at the beginning of each assignment PDF to submit with their code a design which included the Understand the Problem, Devising a Plan, and Testing steps. Though data was not being collected at this point, anecdotally, teaching assistants (TAs) grading the assignment reported that students were either not submitting designs, submitting partial designs missing some of Polya’s steps, or submitting low-quality designs with all the steps. It was believed that this was largely due to the low point value associated with the design as part of the assignment, as well as students writing a design after they had already programmed the assignment. Many students supported this hypothesis by admitting to their lack of engagement or creating the design document after writing the program. It was clear that students were not seeing a benefit in designing as they did not engage properly with the activity and received little enforcement or guidance on the process. This led to mandatory recitations in the first-year CS courses focusing on design concepts and practices.

**Understanding the Problem:**
This problem is asking me to read an unsigned whole number value, n, from the user, and then read n unsigned real numbers, which represent test scores, from the user. These scores need to be between 0 and 100, as well as a valid real number. If the user doesn’t enter a valid number or a number in the range, then an error message is printed, and the user is prompted to enter a new number without taking away from the n valid numbers the user is entering. After the user enters n valid real numbers in the range of 0-100, then the average is calculated and printed to the screen.

I am assuming the number of tests is an unsigned whole number.
I am assuming the test scores can be unsigned real numbers, instead of just integers.
I am assuming that errors in the user input does not count against the n numbers to enter.

**Devising a Plan/Design**

**Testing:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 0</td>
<td>Nothing, just exit</td>
<td>Yes</td>
</tr>
<tr>
<td>n = 1</td>
<td>Error message and re-prompt the user for a good n value</td>
<td>Yes</td>
</tr>
<tr>
<td>n = 1.5</td>
<td>Error message and re-prompt the user for a good n value</td>
<td>Yes</td>
</tr>
<tr>
<td>n = 1</td>
<td>Prompt user for 1 test score</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 1 - Example Design Template

**Understanding the Problem**
In your own words, explain what YOU think the problem is asking you to do. In this section, document your uncertainties about the problem and anything else that you feel was unclear or vague. This is to ensure that YOUR understanding matches MY understanding of the problem.
Hearing a recitation with the course provided a framework for discussing and enforcing design prior to coding, more feedback for design, and encouraged engagement with the activity by requiring a peer-reviewed design document submitted on Canvas (an online learning management system) one week prior to the assignment’s due date. The student’s recitation peer leader would grade and comment on the design to encourage the student to improve their designs over time, and students conducted peer reviews of the designs in their recitation section through Canvas to be exposed to alternative ways of thinking, to constructively provide additional comments, and to enforce taking designs seriously.

With a structured method for examining and enforcing student design in the first-year courses, CS 1 and CS 2, a study was conducted to see what students did when required to provide design documents based Poly’s problem-solving steps, and how instructor guidance in a syllabus can change the behaviour of the students. The study presented in this paper examines 2,797 designs over four 10-week terms with the following research questions and motivations:

**RQ1**: Does the inclusion of categories differ with instructor guidance in the syllabus?

**RQ2**: What are student attitudes toward guidance on designs and their value of design?

### 4.1 Course Structure

Four first-year CS courses from spring 2016 – spring 2017 (referred to as Classes A, B, C, and D respectively) were part of this study. All courses were taught by the same instructor, providing consistency in the implementation of the recitations. The recitations count as 20% of their total course grade and have their own syllabus to provide grading requirements for submitted designs and peer reviews, in addition to the explanation of Poly’s steps presented in Table 1 and the example design document shown in Figure 1 above. The recitation syllabus evolved each term of the study to address peer leader grading confusions and promote student engagement with the activity by adding increased clarification and point values to the syllabus (see Table 2). The changes are influenced by McCracken et al. (1999), who found that students do not inherently know what design is and need to be taught or given clear definitions of design. Even though the students were given a thorough explanation of what was required and an example design document, the researchers believe that increasing clarity of expectations in the recitation syllabus (or grading rubric) leads to better designs.

<table>
<thead>
<tr>
<th>Class</th>
<th>Designs</th>
<th>% of Grade</th>
<th>Expectations of Design According to the Syllabus</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class A</strong>: Spring 2016</td>
<td>5</td>
<td>25%</td>
<td>“Recitation will focus heavily on design concepts. To enhance learning, every student will have to submit their design to Canvas for the current assignment during the first week of the period in which the assignment is assigned… The design does not need to be correct but show good faith effort to creating quality design for the current assignment.”</td>
</tr>
<tr>
<td><strong>Class B</strong>: Fall 2016</td>
<td>4</td>
<td>40%</td>
<td>“The design does not need to be correct but show good faith effort to creating quality design for the current assignment, and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (4 pts), and 3) Test Cases (4 pts).”</td>
</tr>
</tbody>
</table>

### Table 1 - Poly’s problem-solving steps with detailed descriptions

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Understanding the Problem</td>
</tr>
<tr>
<td>2</td>
<td>Devise a Plan/Design</td>
</tr>
<tr>
<td>3</td>
<td>Testing</td>
</tr>
<tr>
<td>4</td>
<td>Review</td>
</tr>
</tbody>
</table>

---

**Notes:**

- The expectations of design have been updated throughout the year to ensure that students are aware of the requirements.
- Each student is required to provide a detailed description of their design process, including any algorithms or pseudocode used.
- Test cases are an important part of the assignment, and students must provide a clear explanation of their testing strategy.

---

**References:**

pts). You will receive one point for each area by turning in the work. The remaining points for each area will be based on how thorough and complete each section is. For example, restating the problem for the design area 1) Understanding the Problem will only get you one point. You must describe and justify your understanding of what the problem is asking to receive full credit.”

<table>
<thead>
<tr>
<th>Class C: Winter 2017</th>
<th>4</th>
<th>40%</th>
</tr>
</thead>
<tbody>
<tr>
<td>“The design does not need to be correct but show good faith effort to creating quality design for the current assignment and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (must contain function details and header info) (4 pts), 3) Test Cases (must contain good, bad, and edges cases) (4 pts).”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class D: Spring 2017</th>
<th>5</th>
<th>40%</th>
</tr>
</thead>
</table>
| “The design does not need to be correct but show good faith effort to creating quality design for the current assignment, and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (must contain function details and header info) (4 pts), and 3) Test Cases (must contain good, bad, and edges) (4 pts). By default, you will receive one point for each area addressed in the design (up to 3 points for just turning in something!). The remaining points for each area will be based on how thorough and complete each section is.

For example, restating the problem for the design area 1) Understanding the Problem will only get you one point. You must describe and justify your understanding of what the problem is asking you to receive full credit, i.e. both points. For test cases, you MUST have good (1 pt), bad (1 pt), and edge (1 pt) cases to receive full testing credit, and your design needs to include details for the logic in the functions (1 pt), as well as information about the pre/post conditions and return values (1 pt), and the relationship among the functions/classes (1 pt) for full design credit.” |

Table 2 - Recitation Syllabus/Grading Rubric Changes Over 4 Courses (A-D)

<table>
<thead>
<tr>
<th>Classes A and D were required to turn in designs for 5 assignments covering the following topics.</th>
<th>Classes B and C were required to turn in designs for 4 assignments covering the following topics.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 2-D arrays and Files</td>
<td>• Repetition</td>
</tr>
<tr>
<td>• Classes</td>
<td>• Functions</td>
</tr>
<tr>
<td>• Inheritance</td>
<td>• 1-D Arrays</td>
</tr>
<tr>
<td>• Polymorphism</td>
<td>• 2-D Arrays</td>
</tr>
<tr>
<td>• Linked Lists</td>
<td></td>
</tr>
</tbody>
</table>

Even though the assignment topics differ in classes A and D from B and C, the research questions in this paper are addressing the inclusion of features in the design document based on the change in instructor guidance given in the recitation syllabus.

4.2 Participants

The researchers obtained consent from students to examine their designs in 4 first-year CS courses from spring 2016 – spring 2017 (referred to as Classes A, B, C, and D respectively). Table 3 shows the consenting population differences across courses. Each course had slightly different consent rates and grade distributions. It is important to note for analysis purposes that most of the participants in this study are “above average” students.
4.3 Classification Categories

Based on Thomas et al. (2014), the researchers developed six categories for classifying what students include in their design documents to evaluate the quality of design (see Table 4). Eighty-six random designs, 10% of the data collected from spring 2016, were used to determine a suitable classification for quantifying the design documents based on an inter-rater reliability (IRR) greater than 80%. Though the rubric proposed by Thomas et al. (2014) had six categories with 0 to 5 representing the level of quality (ranging from informal to expert), the research questions in this study can be answered using a binary value to represent if the design contains certain features. Note that Code Present is seen as a negative feature, while the rest of the categories are positive or neutral. Program design in the context of this study should not be code specific.

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding the problem (UP)</td>
<td>Framed the design and states what the design solves for.</td>
</tr>
<tr>
<td>Relationship Among Parts (RAP)</td>
<td>Provided text or a picture explaining how each function and/or classes relate(s) to each other</td>
</tr>
<tr>
<td>Logic (L)</td>
<td>Provided details for each function and/or class, expect pseudocode or code, with an emphasis on functionality details (.h doesn't count)</td>
</tr>
<tr>
<td>Code Present (CP)</td>
<td>Specified specific syntax.</td>
</tr>
<tr>
<td>Diagram/Picture (DP)</td>
<td>Drew a diagram, picture, or flowchart to represent idea</td>
</tr>
<tr>
<td>Testing (T)</td>
<td>Provided a test plan</td>
</tr>
</tbody>
</table>

Table 4 - Design rubric with >80% inter rater reliability

5 Results

RQ1: Does the inclusion of categories differ with instructor guidance in the syllabus?

Syllabus changes seem to correlate with increased inclusion of some design components by the students. A Kruskal Wallis test was run comparing each design in each term to determine if there were differences in inclusion rate between the terms because the data was not normally distributed. A Pairwise Wilcoxon test was then run to determine which terms of the four differed significantly, if at all. The hypothesis for RQ1 is that increased detail in the design syllabus will lead to more inclusion of the categories.

Understanding the Problem and Testing were included at a higher rate each term, as shown by Figure 2. The inclusion of Understanding the Problem was found to be significantly different each term for each design with a p-value < 0.05. For testing, there were no significant differences in inclusion for the first three designs in courses A and B, but all other classes and designs did show significant differences. This can be directly linked to changes in the syllabus. Recall from Table 2 that the key syllabus change from class A to B was the enumeration of what was required in the design, namely Understanding the Problem, Flowchart and/or Pseudocode, and Testing point values and an increased overall weight of the designs to the recitation grade. The key difference between courses B, C, and D were details for the Flowchart and/or Pseudocode and Testing, which profoundly affected the rate of inclusion for these categories. These results seem to align with other research on motivating student performance using rubrics (McCracken et. al., 1999) (Jonsson and Svingby, 2007). However, there is a plateau effect in
classes C and D that is likely due to many students who take class C also take class D in their first-year, and they may already know what is expected of their designs.

![Graph](image)

**Figure 2: Change in Percent Inclusion of Categories Over Time**

A place where refinement of the syllabus may still need to occur is in the *Relationship Among Parts* and the *Logic* components. There was no significant difference across the designs or terms for the inclusion of *Logic*, except for design 3 where A and B, A and C, and B and D all significantly differed from one another. The *Relationship Among Parts* category showed a variety of significant differences across the designs and terms. Figure 3 shows the percent inclusion for *Relationship Among Parts* over time. Note that even without running statistics, the graph is rather sporadic. When the Kruskal Wallis and the Pairwise Wilcoxon tests were run, it was discovered that there was no difference between A and B on design 1 or C and D on design 1. This could be because the design was the first one of the term and every student was starting in the same place with their understanding of the *Relationship Among Parts*. Course B design 2 was statistically different from all of the other terms. This is potentially due to the increased number of nonmajors in this course engaging with the *Relationship Among Parts*. Design 3 saw a statistically significant difference each term, starting after course B. Design 4 and design 5 saw significant differences each term.

![Graph](image)

**Figure 3: Change in Percent Inclusion of Categories Over Time**

As seen in Figure 4, students are more likely to include diagrams and pictures versus code in a specific language. This might be due to the design document template provided in Figure 1. It is interesting to see that diagrams and pictures decrease in design 5 for class A, while the amount of code increases. In every course you see a downfall in diagrams and pictures included over time, but primarily this happens the most in the last design in courses A and D. This might be due to the topic of the assignment, which is linked lists, but this is a time when students should have an increased number of pictures in their designs. However, most terms, except class A, decrease the amount of code included in their designs over time. This is encouraging, since the last assignment in class A and D are the same, and we do not continue to see the same trend in class D, after the syllabus changes.
The fact that the differences are so sporadic over time indicates that Relationship Among Parts is likely not well defined for the students. They do not know how to include it or address it in their designs. This could be attributed to the overall idea of design still not being clear in the syllabus. When Understanding the Problem and Testing were both given special attention and point values, there was an observable up-tick in their inclusion rates. The design portion of the syllabus likely needs more detail or there need to be more concrete examples for the students. The request for more examples has been common from students, and depending on their recitation peer leader, they may or may not be getting the necessary support they need to design successfully in this area. The later classes, C and D, also saw an emphasis on function details and headers in the syllabus, which may have caused more students to focus on the functions rather than the relationship between the functions.

**RQ2: What are student attitudes toward guidance on designs and their value of design?**

In courses C and D, students were asked to fill out a survey at the end of the course speaking to their experience in recitation where the design work was conducted. In class C, 166 of the 213 consented participants responded. 67% of them wanted more guidance on design and 79% thought that creating design was useful. In class D, 160 of the 177 consented participants responded. 64% wanted more guidance on design and 86% thought creating design was useful. A Kruskal Wallis test was run to determine if there was a significant difference between these percentages. There was no difference between wanting more guidance but there was a significant difference in believing that design was useful. Many of the students who took class C proceeded into class D. The increase value of design corresponds with the established idea that students improve in design over time, likely because they engage with it more as they see value. Students wanting more guidance on the design also supports the acknowledged issue of not enough clarity in the current syllabus on design or not enough concrete examples being provided on what quality design is.

**6 Conclusions and Future Work**

The study sets out to show how instructor guidance can influence what students include in their design documents. Despite being given an example design document and the exact problem-solving steps to include, first-year CS students are motivated by points and specific directions. While the study is unable to present data from before the study began, the results are still valuable. It is known that before requiring design to be submitted one week prior to the coded program, most students would not engage with the activity and would submit mediocre work to get a few points. By shifting the focus of the class to prioritize 8% of the overall course grade to design, more students are engaging with the activity.

The binary scale, while useful to say if desirable artifacts are present, does not quantify the quality of the presented artifact. The Relationship Among Parts and Logic categories need a non-binary scale. The approach in Thomas et al. (2014) was too broad to achieve IRR on the designs presented in this study. The rubric may have worked better if notation was formalized. The taxonomy proposed by Castro
and Fisler (2017) was too closely tied to code structure for the researchers to use in this study; however, the level of detail and the verbiage used may be adaptable to the needs of language agnostic design. At this time, we do not investigate the quality of the design, but we plan to make correlation between the design and corresponding code quality in future studies, which will leverage work presented by Stegeman et. al. (2014). The themes discovered by Yeh (2018) study are very relative to this research and could provide a good rubric for evaluating students’ strategies for devising a plan in their design documents.

The changes demonstrated in the syllabus over time lead to an increase of inclusion of certain categories which were previously being ignored. Now, most students engage with the Understanding the Problem and Reflection stages of the problem-solving process. While the impacts of this engagement are unknown at this time, future work hopes to examine more correlations in regard to code produced and time spent coding or debugging. This study also demonstrates that there is a challenge to teaching design well in the first year. However, improvements can be made between terms, and the way students value design can be changed.

7. Acknowledgements
We will insert upon acceptance.

8. References


Abstract
For decades, the software industry has struggled with change, continually devising new methods to better control and manage the risk to software development projects. This paper attempts to reconsider change as a positive force that can produce better, more resilient software. It argues for providing “users” with greater creative influence throughout the design process, as co-designers; and to support them with material software tools that will allow them to enact unanticipated changes after design is complete.

1. Inward-Facing Software Methods
Change is hard. Much of the history of software design methodology has focused on devising strategies for controlling, minimizing, or formalizing change. Some methods attempt to reduce the possibility of disruption by “getting it right” up-front, via the use of formalized requirements gathering strategies and user research models. Others aim to “embrace change” by establishing management and programming tactics that reduce the cost of responding to changes introduced at any time during the software development process. All these methods invariably look inwards, at the working practices of teams of expert designers and programmers who have some measure of influence over the ways in which change is conceived, managed, and responded to during the process of creating a piece of software.

Iterative, or agile, design and development methods involve the incremental evolution of a software product over the course of many short cycles of research, design, implementation, and stakeholder feedback. This approach offers effective opportunities for technical teams to change direction quickly and to recalibrate a program’s features, user interface, and other designed qualities more quickly and at a lower cost. However, it is notable that many of the most popular agile approaches to software development, such as Lean, Scrum, and others, still retain an explicitly industrialist mindset, often mimicking the assembly line manufacturing processes and production strategies of consumer objects such as automobiles. For example, Toyota’s “lean” Production System is often cited as a significant influence on agile software development methodologies. While most agile processes have a “customer” or “product owner” role within the team, this role is usually performed by a single person who is embedded within or nominated by the management structure of the organization, and who may not represent the diverse needs and perspectives of the day-to-day users of the software.

Here, an acceptance of change means cultivating strategies and technologies that support increased modifiability, parameterization, or reuse of software artefacts during the production process. Industrialized product design methods—including even the most flexible agile practices—still assume a conventional

---

1 As an example, Alan Cooper’s interaction design methods primarily provide design-based management strategies that centre around the creation of formalized design models such as personas and scenarios at the beginning of the software lifecycle. See Cooper, Alan et al (2014). About Face: The Essentials of Interaction Design 4th Edition. Wiley.


3 In his “Agile Versus Lean,” Martin Fowler states that “There was a connection between lean manufacturing and agile software from the beginning in that many of the developers of the various agile methods were influenced by the ideas of lean manufacturing.” https://martinfowler.com/bliki/AgileVersusLean.html Mary Poppendieck has elaborated an entire software development methodology based on the direct application of Toyota’s automobile manufacturing methods to programming, such as in her “Lean Software Development.” https://dl.acm.org/citation.cfm?id=1248986
producer/consumer dynamic, in which an object (in this case, the software product), though it may have changed frequently and freely over the course of its creation, is delivered to a consumer in a finalized and largely unchangeable form⁴.

2. The Politics of Use

For users, on the other hand, change is arguably even harder. Lacking direct influence over the process of its creation, software products tend to be a “take it or leave it” proposition for users—they work as they do, with perhaps limited configurability or the possibility of buying costly custom tailoring provided by vendors or consultants. If an individual user needs something different, often their only recourse is to look elsewhere, at other products. Yet at the same time, change is increasingly imposed on users, with the rise of mandatory software updates and cloud-based software-as-a-service platforms. Beloved features may disappear, move, or be recast by software designers at any time and without notice or permission, leaving users to adapt or relearn their hard-earned workflows.

There is, of course, a power dynamic at work here. While the difficulties of change are felt by software designers and users alike, the power to enact (or forgo) changes, to manage their impacts, scales and timing, rests overwhelmingly in the hands of those who initially created the software. Although users often pay for their software, substantive ownership of it—that is, creative control over what it is, when it changes, and how it is intended to be used—remains too often locked up with those who originate it. This dynamic is reflected in the methods by which much software is made, the models by which it is deployed and distributed, and by structure of many programming idioms themselves—compiled, unidirectional, insular⁵.

It is becoming increasingly clear that equitably-designed software systems—those that are capable of including a diversity of needs and experiences—may never be fully realized using inward, expert team-facing processes that result in static software “products.” Software’s complexity, and the dizzyingly varied, situated needs of different people and communities, suggest that we may never be able to design one single artefact that fits everyone. At the same time, the costs of specialized, isolated, and incompatible systems are increasingly unsustainable⁶. Today’s methods leave designers in a situation where their only substantive means for dealing with change is via the power of omission—intentionally leaving out features that have been measured, through one process or another, as having limited value, scope, or utility to a hypothetical average, norm, or majority of users⁷. For users on the margins, such as those who with disabilities or who have difficulty with literacy of complex digital systems—and thus who may need features that are specialized or individualized in some way—this “economy of the mainstream” perpetuates marginalization and exclusion.

Change is hard, yet we need more of it—in new forms. We need design methods that open up new vectors of change, which include the reciprocal participation of diverse individuals and communities from the very beginning, and which allow those users to continue the design process—configuring, adding and removing features, connecting software artefacts together, sharing customizations with each other—even after the

---

⁴ On the other hand, it is notable that agile methods have flourished within in-house or long-term contracting software development teams, where there is an ongoing relationship between the software’s developers and the organization sponsoring it. This enables changes to be made continually throughout the lifetime of the software’s use within the organization. The scale and cost of this relationship, however, is far out of reach of ordinary users, and is typically reserved for large enterprises who can afford to continuously fund an active software development team.


⁶ The assistive technology market has been plagued by the issues associated with specialized, poorly interoperable solutions, as documented in Treviranus, Jutta (2018). Let’s not spend public funds to perpetuate digital disparity. Medium. Available at https://medium.com/@jutta.trevira/lets-not-spend-public-funds-to-perpetuate-digital-disparity-a4413132ca

originary design process is finished\textsuperscript{8}. By reorienting our design processes around the idea of \textit{continuing creativity}, we raise the stakes for the embrace of change while aiming to rebalance the relational dynamics of software’s design and use. This perspective emphasizes the role of non-expert “users” as co-designers during the initial process of software’s conception and creation, as well as advocating for new programming tools that support the continued redesign of software artefacts by non-programmers, even after they have been designed, coded, and shipped.

3. Inclusive Co-Design
Co-design is designing \textit{with}, not simply \textit{for}. It involves asking the people who might otherwise just be "users," particularly those on the margins of today’s technology experiences, to be part of the design process from the beginning. Lacking the proceduralism of industrialized design methods, co-design typically starts with a process of discovering and negotiating roles—asking participants how, when, and how often they want to be involved, and making space to accommodate different “scales” of investment and engagement. As a result, co-design takes time to do well. Its processes need to be tailored to the unique context and situation that a design intervenes into, and it demands that all participants have equal access to the information—plans, ideas, prototypes, and works in progress—that is essential for full decision-making and responsible contribution. A starting point for this involves an opening up of agile’s iterative and incremental processes to be more porous and include a broader range of team participants and modes of engagement. An example of this is the Fluid Project’s co-design practices, which combine designing and planning in an open wiki, remote participation in design crits and decision-making processes, and “embedded” co-design activities, all within the context of an open development community\textsuperscript{9}. Fluid’s embedded co-design process involves creating toolkits of activities and resources that are intended to help engage people in the design process, situated within the context of their own communities, participants, and places. These toolkits are given to communities to organize and use themselves, without Fluid’s designers present, along with training and mentorship if needed. Thus, multiple levels of co-design can be performed independently and at different locations without central organization, all facilitated by those who are trusted within the communities they are practicing within.


Co-design must be reciprocal. As Sherry Arnstein notes, participation is power\(^\text{10}\). Co-design is not a case of designers allowing users to participate, but rather, fully engaging them as citizens of the technological spaces that they are increasingly inhabiting, working, and expressing themselves within. This involves practicing in ways that are self-aware of the profound power and privilege that technologists hold, and in finding ways to fully share and give up that power. It is not enough simply to ask people for feedback; participants in co-design need to know that their input and ideas can have real power and influence in the resulting software. Co-design demands the knowledge that ideas will be heard, that they can have a direct influence, and that the mechanisms and processes by which they will potentially be enacted are clear and accountable. To this end, the Co-Designing Inclusive Cities project is currently developing a toolkit of co-design practices and activities, along with community-led measures that can be used by participants to assess the extent and effectiveness of their engagement\(^\text{11}\). The project’s toolkit will be used to support the community-led design of civic infrastructure and connected cities such as Toronto’s Quayside project—an area in which there is currently significant risk of placation and consultation tokenism, and thus co-design practices are acutely needed.

Co-design is not a new approach. It draws its roots from Scandinavian cooperative design projects such as Utopia, in which labour union members directly contributed to the design of new computer workstations\(^\text{12}\).

---

10 “Citizen participation is a categorical term for citizen power. It is the redistribution of power that enables the have-not citizens, presently excluded from the political and economic processes, to be deliberately included in the future. It is the strategy by which the have-nots join in determining how information is shared, goals and policies are set, tax resources are allocated, programs are operated, and benefits like contracts and patronage are parcels out. In short, it is the means by which they can induce significant social reform which enables them to share in the benefits.” Arnstein, Sherry. (1969). “A Ladder of Citizen Participation.” AIP, Vol. 35, No. 4, July 1969, pp. 216-224.


Elsewhere, Elizabeth B.N. Sanders has emphasized the value of engaging participants creatively within what she calls the “fuzzy front end” of design—the early, ambiguous, and chaotic phase of the design process prior to a product being fully conceptualized\(^{13}\).

While co-design clearly opens new possibilities for more equitable participation by workers and citizens within software design teams during the scope of initial creation, ongoing change remains a factor that needs to be addressed. People change over time, our needs change, and unanticipated possibilities invariably emerge from use and experience. This suggests, then, that there is a need to find ways to extend the creative participation that co-design offers early in the process, continuing it even after the initial design process is finished.

4. Material Systems

Mads Dahlke is the Danish host of a popular do-it-yourself YouTube channel called Sail Life\(^{14}\). For the past several years, Dahlke has documented his process of taking a fully functional, thirty-year-old sailboat, disassembling it in a variety of artful and intrusive ways, and rebuilding it to suit his own needs and tastes as a sailor with ambitions of crossing oceans in it. Among other projects, Dahlke has removed and replaced the boat’s entire deck, designed custom-fit fuel tanks, completely reconfigured the layout of a cabin to better suit his needs as a workspace, and repaired many flaws resulting from oversights or cut corners during the boat’s original design and construction. Indeed, watching Dahlke’s weekly uncovering of new issues and challenges suggests that the boat’s original designers simply never conceived that their product would still be in active use today, nor did they design it with any intention of it being modified in the ways that Dahlke has accomplished. Yet Dahlke admits he is not a professional boat builder or expert restorer. He has pursued his project by acquiring some generalized technical skills and commodity tools, while participating in a larger community of other sailors and do-it-yourselfers who have worked on similar projects and shared their own learnings.

Although Dahlke’s endeavour may not be particularly unique when seen from the perspective of DIY repair and renovation, it represents something that is very difficult, if not impossible, to do with software systems today. Coincidentally, Dahlke’s “day job” is as a professional software developer. And yet, despite his elite technology skills, Dahlke would nonetheless be hard-pressed to perform a comparable series of transformations and personalizations on a (closed source) piece of software that someone else had designed before him—especially at a comparable cost. His power to change his sailboat after the fact of its design far exceeds his power to change software under similar circumstances. For a non-professional—an “ordinary user”—such transformations would certainly be out of reach entirely. The cost and complexity of unanticipated change in software is always significant\(^{15}\), often intractable, and even sometimes inconceivable\(^{16}\).

Material software is software that provides the power to be adapted, configured, re-presented, augmented, or separated in various ways, without needing to have been part of the original software development process or to have access to “elite-level” programming knowledge or tooling. In its simplest form, this materiality may take the form of simple transformations of a software’s user interface, such as those provided by the UI Options accessibility preferences framework\(^{17}\). Fluid Infusion (of which UI Options is

---


14 Sail Life. (n.d.) Home [YouTube Channel]. Retrieved from https://www.youtube.com/channel/UC5xDbh2bPWNwDvI99P0mgA

15 On the order of 80% of software development costs are related to maintenance due to changing requirements. See Kniessel, et. al. (2002) “Unanticipated Software Evolution.” In J. Hernandez and A. Moreira (Eds.): ECOOP 2002 Workshops, LNCS 2548, pp. 92–106.


17 https://build.fluidproject.org/infusion/demos/prefsFramework/
a part) is an effort to build a software framework that supports an open authorial ecosystem in which any software expression can be modified, refined, and replaced by consecutive authors, without breaking the linkages and connections amongst this network of expressions\textsuperscript{18}.

Material metaphors for software, of course, have distinct limitations. Software, we might imagine, is a “material of the mind,” yet one which is expressed in a uniquely computational form—infinitely reproducible, demanding of precision and detail. At least in theory, highly pliable. And yet anyone who has, for example, diligently attended to a complex writing project knows that this may not be literally the case. Ideas, in system, are complex and even sometimes inconceivable outside of the context in which they were originally situated and elaborated. Software is not yet, and perhaps never can be, a craft\textsuperscript{19}. Its material qualities may be far too different, ultimately more subject to change and systematic contingencies than artefacts in the physical world. Nonetheless, these metaphors from the physical and craft worlds may help us to see more clearly the potentialities in the medium that we have overlooked or hidden away in computation’s dominant formalistic and neo-Romantic creative constructions.

5. Conclusion
So how can we start to come to terms with change, to give it space within our software designs as an opportunity, not just a risk? There are, of course, no silver bullets; no easy-to-follow checklists or master methodologies that will produce software that has the resilience, flexibility, and longevity that is needed to meet and include users where they are, rather than continuing to force them to adapt and compromise. Two interrelated strategies, described here as continuing creativity methods, suggest potential ways to fully embrace change. First, narrowing the gap between use and design via co-design, particularly by engaging diverse users as equals in the process from the beginning. Secondly, creating software tools, programming frameworks, and authoring environments that will increasingly support users in modifying and redesigning them, even after the software has shipped. These approaches represent an early and speculative collective work-in-progress, which will undoubtedly benefit from continued experimentation, exploration, mistake-making, and active participation by “users,” designers, and programmers alike.
