appropriate. Hopefully the beneficiaries of such research are appreciative, but we suspect that the motivations of the researchers may be in part the opportunity to design programming languages that can be evaluated according to different sets of criteria from those that are over-familiar in computer science.

An important characteristic of end-user programming research is that end-user programmers should not be regarded as "deficient" computer programmers, but recognised as experts in their own right and in their own domain of work. They might only write programs occasionally or casually, but it is possible that they have done so for many years, possibly distributing their work for use by many others. From this perspective, research into the programming behaviour of first year computer science students, although convenient and commonplace, provides little relevant insight into the needs of end-user programmers (Blackwell, in press). Similarly, attempts to investigate "natural" programming concepts, by studying school children before they have encountered any other language (Pane et.al. 2001), are of great interest to researchers, but may not be directly relevant to the needs of expert end-user programmers.

The real benefit in studying unusual populations of programmers, whether they are representative of end-users or not, is that in addressing more unusual needs we may find more creative solutions. In fact many of the greatest adventures in programming language design arose specifically from encounters with new user populations. Kay's work on Smalltalk was motivated by the needs of children (Kay 1972), as was Kahn's "programming as videogame" system ToonTalk (Kahn 1996). The spreadsheet was invented in response to the needs of business school students (Power 2004), and our own more modest work leading to the unusual tangible programming language MediaCubes was a response to the problem of configuring networked home appliances (Blackwell & Hague 2001).

One substantial advantage of these unusual user populations is the way in which the resulting inventions are inspired by very different contexts of programming activity. For example, the context of the home is rich in opportunities for programming, but these are dominated by social interactions between family members (Rode, Toye & Blackwell 2004). Consideration of the purpose of programming in schools leads not only to innovations for individual students (Blackwell 2003), but also imaginative response to the school curriculum (Rode, Stringer et. al. 2003) and even new teaching models as when students from one year act as apprentices to mentors who took the same course in the previous year (Ching 2000).

This research strategy forms the focus for the rest of our paper. We consider a new domain for research in terms of i) the ways in which these users' tasks are unlike the normal models of program execution, ii) the ways in which these users' needs are unlike the needs of professional software developers, iii) the ways in which these users' needs are unlike previous research into end-user programming languages, and iv) the ways in which the context of use may inspire novel conceptions of the nature of programming.

2 Music and Programming Interfaces

The domain of music technology provides a exciting range of challenges and analytic perspectives both for HCI and for Psychology of Programming. Traditional musical instruments include highly evolved user interfaces, often addressing issues that are of pressing concern in contemporary HCI. Most instruments offer sophisticated modes of bimanual interaction (MacKenzie & Guiard 2001). Instruments like the concertina demonstrate how a user interface can be based on an elegant visualisation of abstract musical structure, integrating melodic (the tune from note-to-note) and harmonic (chords to go with those notes) into a two-dimensional layout of controls (Holland 1994). On the other hand, instruments like the bassoon provide us with a puzzle that challenges conventional ideas of usability, inconsistent with the standardised key layout of other instruments, and with peculiarities such as the fact that different keys should be pressed to produce the same note, depending on how hard the player is blowing (Derrett 2004).

Written music notations have many features in common with programming languages, especially when analysed in terms of the Cognitive Dimensions of Notations framework. Our study of a music typesetting system, comparing it to a range of programming languages, found that Cognitive Dimensions could express concerns common to both domains (Blackwell & Green 2000). In this case, the users of music notation systems are composers and editors of music, rather than musicians – a distinction that, as we shall find, is far less clear-cut in the case of live coding.

Electronic music technology has also introduced a significant engineering element into the production of music, so that recording and post-production studios, as well as performance venues, have a substantial amount of notational content involved in the configuration of electronic equipment (Blackwell, Green &

Nunn 2000). Even music consumers may find that domestic audio equipment includes significant programmable functionality (Blackwell, Hewson & Green 2003). These facilities too, have made their way into performance contexts, both in the development of audio processing software designed for use in live situations (e.g. Ableton Live) and in the appropriation of audio playback technologies like the turntable as new musical instruments (Smith 2000).

The distinction between composition and performance, or between notation and instrument, is becoming increasingly blurred in contemporary music technologies. From the perspective of psychology of programming research, this is a provocative development, because it echoes the way in which programmability is pervading the user interface (Blackwell 2002). In previous work, we have described the cognitive effects of this transition in terms of the Attention Investment model (Blackwell & Burnett 2002). Programming-like interaction techniques similar to macro recording can be seen in many aspects of live sampling and sequencing, as well as in the advanced features of research prototypes such as the D'Groove digital haptic turntable, which can be used to control digital audio files as though they were vinyl records being manipulated by a scratch disc jockey (Beamish et. al. 2004). When we look beyond the individual interaction paradigm to collaborative technology use, live performance often incorporates an astonishing array of programmed beats, notated music played by classically trained instrumentalists, and traditional folk music acquired through instrumental apprenticeship or ethnographic research (all of which can be found in most pieces performed by popular Cambridge band Horace X).

3 Laptop and Live Programming History

Laptop music is not a genre but a characteristic of contemporary performance practice in electronic music, born of the affordability of easily transportable computer systems powerful enough for real-time signal processing. The Austrian collective Farmer's Manual are often vaunted as the first true laptop ensemble (they started performing in 1996), though the use of laptops for digital music performance has been practised since the early 90s, particularly in Japan (Loubet 2000). Unsurprisingly, live electronic music has a heritage far longer than that of the laptop through bulkier apparatus such as IRCAM's 4X or earlier modular synthesis systems like the Sal-Mar Construction, and Atari ST and Amiga computers were sufficiently powerful and portable to enable their use (with MIDI software, 8-bit audio sampling tracker programs or early VJ graphics applications) in late 80s raves.

Whilst many interaction peripherals may form part of the laptop musician's interface, the (much criticised) typical performance mode consists of a single user, interacting via mouse with a GUI-based program, at a gestural rate divorced from the rate of output events, so that causes are uncorrelated from effects. Notwithstanding this basic image, laptopists very much vary in their choice of programs, interface and musical output. Laptops are now a staple of the music scene, whether it is Matmos accompanying Björk, Fennesz live sampling and processing guitar, or extreme sound artist Merzbow building a wall of noise.

The degree of challenge and flexibility in programming music software can be characterised along various continua. Popular live laptop music programs like Ableton Live and Reason offer some sequencing, triggering and processing controls in rigid interfaces, but do not have the algorithmic manouverability and customisation potential of graphical programming packages like Cycling 74's Max/MSP or Miller Puckette's PD. Yet more difficult to master, but with compensatory exploratory potential, come textual programming languages for audio like SuperCollider (McCartney 2002) or ChucK (Wang and Cook 2003).

Live coding (Ward et al. 2004, Collins et al. 2003, Collins 2003) was born out of the possibility of programming on stage with interpreted languages. A few pioneers used FORTH and Lisp in the 80s, and in current practice many different languages are exploited, some original and devised for live coding applications. The most widespread are probably the aforementioned SuperCollider, which is a Smalltalk derived language with C-like syntax, and most recently ChucK, a concurrent threading language specifically devised to enable on-the-fly programming. Adaptations of conventional programming language environments are also extant, for example Alex McLean has written his own customised text editor for Perl with cued or looping interpretation (McLean 2004).

Historically, the first known live coding performance is that of Ron Kuivila in 1985 at the Amsterdam based music research institute STEIM, on a desktop computer. Somewhat anticipating later developments, his half hour FORTH performance ended with a system crash. The Hub, notable as the first computer

network band, were also active in the late 80s, often programming new processes during performance, though this was not made an explicit part of the act. The audience were free, however, to wander amongst the group observing their activities. A second wave of live coding began around the year 2000 with laptop performers following Julian Rohruber's experiments with SuperCollider, including his Just in Time Library for performative code, and the live shows of custom software laptop duo slub, who followed a mantra of 'project your screens' to engage audiences with their novel command line based music programs. Recent years have seen further expansion of live coding activity, and the formation of an international body to support live coders- TOPLAP (Ward et al. 2004). The toplap.org site and mailing list is the most active current home for this artistic practice, and TOPLAP have been booked for such electronic music festivals as Ultrasound 2004 (Huddersfield), transmediale 2005 (Berlin) and sonar 2005 (Barcelona).

4 Live Programming Technology

In order to focus on the issues of user interaction with laptop and live programming software, we offer a brief analysis comparing the Ableton Live sequencer to the ChucK on-the-fly programming language in terms of some of the Cognitive Dimensions of Notations (Green & Petre 1996). Screenshots of the two basic interfaces are provided in figures 1 and 2 respectively, and names of Cognitive Dimensions (CDs) in our analysis are italicized in accordance with usual convention.

It is immediately apparent that Ableton is a colourful and attractive integrated GUI application, of a style that is typical both of multimedia production software and of recent generations of laboratory instrumentation systems. In contrast ChucK is invoked as a relatively intimidating command line executable, for which input code must be written in a separate text editing program. Ableton offers high *visibility* of available operations, utilising well known music technology paradigms of the virtual mixer, time (x) against track (y) event sequencer, with MIDI piano roll and audio waveform editors. The operations available in ChucK are only accessible via a separate web page documenting what possible instructions can be typed on the command line and in the programming language files: with a constant need for a newcomer to reference the manual, it has very poor *closeness of mapping*, and terrible *role-expressiveness* because all operations are presented identically, regardless of musical function.

Nevertheless, the predefined (*abstraction hating*) interface of Ableton makes specific assumptions about the music it will treat. The default rhythm (120 beats per minute with a 4/4 time signature) is typical of disco music, and enables rapid setup for the target end-users; mainstream dance DJs. This is not to say that the tool cannot be used for the playback and layering of other audio signals, independent of their rhythmic structure, and then exploited as a processing engine – Ableton supports various third party audio digital effects plug-ins, brought in within a consistent interface. But the *closeness of mapping* to conventional audio processing equipment that is exhibited by Ableton is indicative of a corresponding reduction in potential for creative exploration. ChucK allows the programmer to define their musical representations, within the purview of some minimal time scheduling language primitives, showing no immediate *closeness of mapping* but an *abstraction hungry* system resulting in *hard mental operations* and mastery (certainly for live operation) of many terms.

Fig. 1. Ableton Live software interface

On the dimension of *provisionality*, Ableton can offer immediate gestural rate control via the definition of MIDI mappings and shortcut key commands for toggling mixer states. These allow the live performer to adapt sound immediately and continuously. In fact, ChucK can do the same; but these mappings must be explicitly laid out as code defining the control flow, whereas Ableton has a simple set-up mode for mapping MIDI and keyboard controllers. Neither program can be said to offer simultaneous control of multiple elements without such mappings, for Ableton is otherwise bound to the click and drag paradigm, whilst ChucK just involves typing without any mouse use at all. An interpreted code line, however, can have general consequences for many parameters at once, whereas Ablteon has not even the simplest macro facility.

Commitment to action is immediate in Ableton since all controls have direct consequences, and this may impose *premature commitment*. However, DJ preview of audio tracks can be accomplished for consequence free experimentation and associated *progressive evaluation* before committing to a mix, if the user has appropriate spare outputs from their laptop. ChucK has a more complex *provisionality*. For code, the time of interpretation of a code file can be decided by the user, though one might set-up automated intepretation as an additional constraint. The consequence of running code for *progressive evaluation* is much more difficult to assess: whilst the programmer may claim a good idea of the algorithms, as for any programming activity, the cycle of debugging is there, and is much constrained in live performance. As when Ableton is used without preview audio outputs, the aural result can easily be too loud, timbrally inappropriate, or result in a multitude of other specific musical errors.

The running state of ChucK is fed back through command line text, or through scanning the program code you have written, with a high memory load. Ableton shows much simpler progressive evaluation and memory requirements, due to its reduced core functionality and small number of interface screen sets. With such differences in their difficulty level for live manipulation, ChucK is *error-prone*, whilst the Ableton user can always see the likely scope of their actions given any real familiarity with the program.

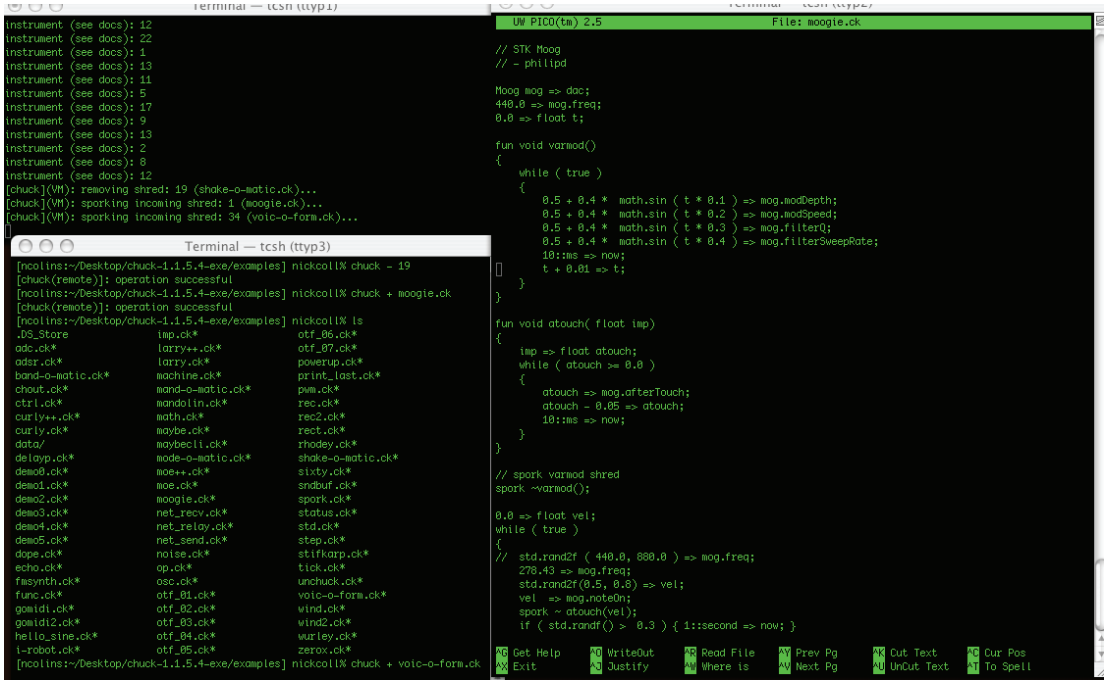

Fig. 2. ChucK command line interface

The reader may therefore wonder why any live performer would choose such a challenge as ChucK when set against the comfortable ride offered by Ableton. An aesthetic response would be to embrace the challenge of live coding; the virtuosity of the required cognitive load, the error-proneness, the diffuseness, all of these play-up the live coder as a modern concerto artist. But a key concern remains the representational paucity of programs like Ableton, which are biased towards fixed audio products in established stylistic modes, rather than experimental algorithmic music which requires the exploratory design possibilities of full programming languages.

5 Task Demands of Live Programming

We wish to consider live programming, not simply from a descriptive perspective, but from a design perspective. We ask the motivating question: What kinds of tool might be required in future to support the practice of live programming, and how would the design requirements for such tools differ from those in other end-user programming tasks? A clear utilitarian design focus like this helps us to see beyond a potentially narrow focus on the tasks of coding (as practitioners have chosen to describe their own work), to tasks that are analogous to every aspect of the software engineering process. These might include requirements analysis, design, reuse, debugging, maintenance and so on. This research strategy is closely related to the work of the EUSES consortium, which places its research emphasis not simply on end-user programming, but on end-user software engineering (Burnett, Cook & Rothermel 2004).

We do not wish, however, to say that live coding performers should work like engineers, simply that their practice as musicians can usefully be analysed by contrast to the practice of professional software engineering. This is slightly different to the justification of the EUSES project, which is concerned with the ways in which much software produced by end-users is deficient (has bugs, does not effectively reuse earlier code, is not documented and so on). There is a clear economic argument for benefits of the EUSES research in terms of “improving” the performance of end-user programmers and making them more like professional software engineers. However in the domain of music, it is not appropriate to assume a deficit model of live coders by comparison to software engineers. Instead, we must make a comparison between the professional practices of software work, and the professional practices of musical work. The remainder of this discussion is structured accordingly.

5.1 Requirements Analysis

Conventional software engineering always starts with a (more-or-less) explicit statement of what the program should do, usually negotiated with the person who is paying for the work. This is seldom the case with music. The economy of music and art production often involves retrospective payment, in which the artist is rewarded for production of a piece according to audience approval. In this case, the artist must anticipate the taste of the audience if he or she wishes to be paid. Alternatively, many musical and art works are produced to commission, but such commissions are based on the previous body of work by the artist, rather than a strict statement of the patron's requirements for the commissioned work. The artist is expected to produce work that is consistent with their previous work, perhaps guided by the patron, but also displaying some degree of creative interpretation and hence freedom from strict control.

In the case of laptop performance, an appearance at a particular event may be commissioned by a promoter, on the assumption that the work produced will be consistent with previous performances. The actual pieces performed will be novel, however, and both promoter and audience expect a unique performance, by analogy to other musical genres in which live performance is expected to be a unique interpretation, even if delivered from the score of a standard work. The relationship between performance and score is a complex phenomenon in the sociology and economics of music. Classical audiences pay both to hear a performance of a particular work, with an agreement or "requirement" that the performers play correctly from an exact copy of the score. However they also pay to hear a particular group of performers, with a requirement that their work should differ in a recognisable manner from other interpretations of the same score.

We see that live programming is very different, in quite illuminating ways, from the treatment of requirements in software engineering. The program is linked to the identity, personality and skill of the programmer in a way that is unusual in other types of software. We suggest that this interesting property may be true of other kinds of end-user programming too. Furthermore, the distinction between notation and performance in music suggests a view of programming in which program behaviour should not be fully predictable, but may vary according to human and aesthetic dynamics in the context of execution. This too may be true of other forms of end-user programming.

5.2 Design

The design phase of software engineering involves the creation of information structures that will serve as a basis for later coding. The possible range of structures is unlimited, but in practice, good structures follow conventions such as structured or object-oriented design. The structures in music are even more closely determined by cultural, perceptual and cognitive precedents. The ubiquitous structures of music are derived from determinants such as auditory scene analysis in the human auditory system, vocal production and linguistic syntax, bodily rhythmic patterns, as well as cultural and genre conventions such as pitch structures, chorus response and dance steps.

When working within the framework of a musical genre, as in much popular laptop performance, the potential range of decisions with regard to the structure of the music is thus somewhat limited when compared to the structure of other software applications. Design notations for music do not need to support arbitrary restructuring of the kind enabled by UML, but may reflect the conventional structure of music notations such as staff notation, chord tablature, or multi-track recording controls. It is an interesting question whether some software structures (recursion, conditional branches) may be adopted in future as part of the conventional listening repertoire for live programming audiences. If this were to happen, then musical notations might evolve to support them.

5.3 Coding

The working habits of the composer are unlike those of the professional programmer, and the work of the musical performing artist is even more unlike programming. However end-user programmers also work in a different way to professional software engineers. Is it possible that end-user programmers might be better understood by analogy to live programming and to musicians, than by analogy to professional software engineers? We note that musicians have a very tight "feedback loop", constantly listening to the results of

their decisions. This is true of most composers, who seldom write directly onto a score, but have a musical instrument close at hand in order to try out ideas. It is even more true of performers, who continuously adjust their playing according to the sound they hear and response of the audience (with the possible exception of British Eurovision contestants!).

The ability of a programming tool to support this kind of feedback is described as the CD of *progressive evaluation*. It is found in interpreted languages where the effect of any command can be tested immediately, and such languages (BASIC, LOGO, FORTH) have always been designed with a special view to use by end-users. Interpreted languages are also popular for use in live programming, for obvious reasons. Two interesting research questions arise. Firstly, do the tradeoffs associated with progressive evaluation impair or obstruct live programming performers in some way? Secondly, can we draw analogies from live programming performance to styles of software engineering such as eXtreme Programming in which rapid feedback from an "audience" and "accompanist" are central to the technique?

5.4 Project Management

The working habits of musical composition and performance appear very different from the ways that professional programmers are managed. However musical work may not be so different to the real ways that programming is done by end-users, or even by the professional programmers of the future. Noble and Biddle's "Notes on Postmodern Programming" (Noble & Biddle 2002) describes a style of programming work that appears to have far more in common with music and musicianship than with conventional assumptions of software project management. Noble and Biddle describe programming as creativity, as performance, as striving toward an undefinable product, fragmentary and abstract, free from narrative, constructing the final work by scavenging through the scrap-heap of the Internet.

5.5 Reuse

In the context of live programming, existing bases of code by the performer and by others are an extremely valuable resource. Fragments of code are assembled like jazz licks or scratch samples. The resulting borrowings of musical intellectual property are so ubiquitous that the critical vocabulary and economic context of music production must constantly describe and attribute ownership. In contrast, the early traditions of software production are more closely based on single-authored literary works. This is clearly inappropriate in the context of some open source software development, and probably even less appropriate for the work of end-user programmers, who often borrow and adapt samples of programs created by friends and colleagues. End-user programming could benefit greatly from closer attention to the way that musical components are assembled for new audiences.

5.6 Debugging

An error in the performance of classical music occurs when the performer plays a note that is not written on the page. In musical genres that are not notated so closely (jazz, blues or rock, among many others), there are no wrong notes – only notes that are more or less appropriate to the performance. Live programming includes notation, but the notation is "performed" automatically by the computer, without error. Should the live programmer be regarded as a composer (whose work may be unconvincing, but not wrong), or an improvising performer? Separation of intent from serendipity is resisted in most performing arts, especially where skilled performance depends on automatic actions too rapid for conscious intent to be articulated.

These characteristics may also be shared by end-user programmers in other domains. Rather than refining and "replaying" informal or casual programs, an end-user programmer may well prefer to accept the results of an imperfect execution. The end-user might perhaps compensate for an unexpected result by manual intervention (like a guitarist lifting his finger from a discordant note), or even accept the result as a serendipitous alternative to the original note.

5.7 Documentation

Programmers are seldom happy at having to document their code, just as musicians seldom like to explain their work. “Writing about music is like dancing about architecture – it’s a really stupid thing to want to do” (Costello/White 1983). Many programming methods have attempted to transform or eliminate the need for documentation, which is imposed on them by the institutional demands of code that must be maintained by other people, delivered to clients, or used by non-programmers. In the case of end-user programmers, none of these things are necessarily true. However live coding provides an interesting contrast in this respect. Laptop performance takes place on stage in front of an audience, and many musical audiences expect not only to hear the music, but see how it is produced. The screen of the laptop is usually turned away from the audience, but a video output from the screen can easily be projected for the audience to view.

Projected laptop performance such as the work of slub, Amy Alexander and other TOPLAP artists does indeed offer views of the code to the audience. In this context the audience themselves, rather than the programmer, might be regarded as the “end-user”. The audience are not producing the code, but they are consuming it. But without knowledge of the language, their consumption even of executable code can be considered as *secondary notation*. This is an unusual perspective from which to view code documentation, but one that may become increasingly common in fields where descriptions of software artefacts are shared between non-programmers.

5.8 Comprehension and Problem-Solving

Much work in the psychology of programming has focused on the critical question of comprehension. In order to write code, the programmer must be able to read it. This happens not only in learning to program from books, or in maintaining code written by others, but even in working with one’s own code, which involves a continual cycle of production and comprehension (Green, Bellamy & Parker 1987).

Live programming performers face the usual problems of code interpretation – identifying meaningful beacons (Wiedenbeck 1986), or locating code within large libraries (Rodden & Blackwell 2002), but they work with significant additional challenges. They must comprehend, adapt and use code in lighting conditions that make it difficult to use paper helper devices, documentation or manuals. Sound levels and audience participation may impose additional cognitive load or impair reasoning, while social expectation of alcohol consumption by performers almost certainly results in the latter. None of these factors are typically found in the work of professional programmers, but once again, may well be far more common among end-user programmers in a variety of contexts.

5.9 Maintenance

If live coding is an ephemeral product that is tied to a specific venue, audience, time or atmosphere in the same way as any other musical improvisation, then live programmers might be considered most fortunate among programmers, in that they never have to do maintenance work! However most musical genres have pursued technologies for preservation of the ephemeral experience, and live coding is not exempt. Perhaps audio CDs or DVDs of AV footage might be sold as keepsakes to committed fans, but would such fans also wish to preserve the code that produced that experience? Code is certainly more preservable than the specific tactile sensations, motion patterns and mental states of musicians improvising on acoustic instruments. Time stamped keystroke data and dribble files could easily be recorded, and might provide data not only for the enthusiastic fan, but for other musicians wishing to quote, extend or appropriate material, in the same way as digital sampling of audio material has resulted in whole new musical genres. Maintenance requirements have therefore been considered by some live coders (for example Julian Rohrhuber in JITLib (Collins, McLean, Rohrhuber and Ward, A. 2003)), particularly when their work extends into the sphere of exploratory programming rather than real-time performance.

6 Conclusions

Live coding is a fascinating and distinctive variety of end-user programming. In the interests of understanding programming from a perspective that is very different from professional programming, it is particularly valuable. Furthermore we do not have to start from scratch in observing and analysing this novel technological context. Music technology has been studied for centuries, and the cultural, cognitive, social and economic factors in music production (especially around notation use and performance) can be analysed from a solid theoretical basis.

Analysis of live coding as a context for programming allows us to escape the implicit assumptions of the commercial office environment in which so much end-user programming has been studied. The programming environments of the future, with increasing deployment of ubiquitous computing technologies, will probably be unlike offices in many ways. We can prepare for this future by studying extreme (as opposed to eXtreme) varieties of programming today. Live coding is thus an ideal research opportunity for psychology of programming.

References

Beamish, T., Maclean, K. and Fels, S. (2004). Manipulating music: multimodal interaction for DJs: Proceedings of CHI'04, pp. 327-334

Blackwell, A.F. (2003). Cognitive dimensions of tangible programming techniques. In Proc. First Joint Conference of EASE & PPIG, pp. 391-405.

Blackwell, A.F. (2002). What is programming? In Proceedings of PPIG 2002, pp. 204-218.

Blackwell, A.F. and Burnett, M. (2002). Applying Attention Investment to end-user programming. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 28-30.

Blackwell, A.F. (in press). Psychological issues in end-user programming. To appear in F. Paterno and H. Lieberman (Eds.), End User Development. Kluwer Academic.

Blackwell, A.F. & Green, T.R.G. (2000). A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell & E. Bilotta (Eds.) Proceedings of the Twelth Annual Meeting of the Psychology of Programming Interest Group , 137-152.

Blackwell, A.F., Green, T.R.G. & Nunn, D.J.E. (2000). Cognitive Dimensions and Musical Notation Systems Paper presented at ICMC 2000, Berlin: Workshop on Notation and Music Information Retrieval in the Computer Age.

Blackwell, A.F., Hewson, R.L. and Green, T.R.G. (2003) Product design to support user abstractions. In E. Hollnagel (Ed.) Handbook of Cognitive Task Design. Lawrence Erlbaum Associates. ISBN 0-8058-4003-6, pp. 525-545.

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

Burnett, M., Cook, C., and Rothermel, G. (2004). End-user software engineering. Communications of the ACM, 47(9), 53-58.

Ching, C.C. (2000). Apprenticeship, Education, and Technology: Children as Oldtimers and Newcomers to the Culture of Learning through Design. Unpublished PhD dissertation, UCLA.

Collins, N. (2003). Generative music and laptop performance. Contemporary Music Review. 22(4), 67-79.

Collins, N., McLean, A., Rohrhuber, J., and Ward, A. (2003). Live coding techniques for laptop performance. Organised Sound, 8(3), 321-330.

Costello/White, T. (1983). A man out of time beats the clock. Musician magazine No. 60 (October 1983), p. 52.

Hartree, D.R. (1950). *Calculating instruments and machines*. Cambridge University Press.

Derrett, N. (2004). Heckel's law: conclusions from the user interface design of a music appliance—the bassoon. Personal and Ubiquitous Computing 8(3-4), 208-212.

Green, T.R.G., Bellamy, R.K.E. and Parker, J.M. (1987). Parsing and gnisrap: A model of device use. In G.M. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, NJ: Ablex, pp. 132-146.

Green, T.R.G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' approach. Journal of Visual Languages and Computing, 7,131-174.

Holland, S. (1994) Learning about harmony with Harmony Space: an overview. In Smith, M. and Wiggins, G. (Eds.) Music Education: An Artificial Intelligence Approach. Springer Verlag, London.

Horace X. website at http://www.horacex.com/

Kafai, Y. B., Ching, C. C., & Marshall, S. (1997). Children as designers of educational multimedia software. Computers and Education, 29, 117-126.

Kahn, K. (1996). Seeing systolic computations in a video game world. Proceedings IEEE Symposium on Visual Languages. Los Alamitos, CA: IEEE Computer Society Press, pp. 95-101.

Kay, A.C. (1972). A personal computer for children of all ages. In: Proc. of the ACM National Conference.

Loubet, E. (2000). Laptop performers, compact disc designers, and no-beat techno artists in Japan: music from nowhere. Computer Music Journal, 24(4), 19-32.

MacKenzie, I.S. and Guiard, Y. (2001). The two-handed desktop interface: Are we there yet? Extended Abstracts of CHI 2001, pp. 351-352.

McCartney, J. (2002). Rethinking the computer music language: SuperCollider. Computer Music Journal, 26(4), 61-68.

McCracken, D.D. (1957). *Digital computer programming*. Wiley.

McLean,A. (2004). Hacking Perl in nightclubs. perl.com article. http://www.perl.com/pub/a/2004/08/31/livecode.html

Myers, B.A. (2002). Towards more natural functional programming languages. In Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, p. 1.

Noble, J. and Biddle, R. (2002). Notes on postmodern programming. In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 49-71.

Pane, J.F., Ratanamahatana, C.A. and Myers, B.A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies 54(2), 237-264.

Power, D.J. (2004). A Brief History of Spreadsheets. DSSResources.COM, World Wide Web, version 3.6, 08/30/2004 downloaded from http://dssresources.com/history/sshistory.html.

Rodden, K. and Blackwell, A.F. (2002). Class libraries: A challenge for programming usability research. In Proceedings of PPIG 2002, pp. 186-195.

Rode, J.A., Stringer, M., Toye, E., Simpson, A.R. and Blackwell, A. (2003). Curriculum focused design. In Proceedings ACM Interaction Design and Children, pp. 119-126.

Rode, J.A., Toye, E.F. and Blackwell, A.F. (2004). The Fuzzy Felt Ethnography - understanding the programming patterns of domestic appliances. Personal and Ubiquitous Computing 8, 161-176.

Smith, S. (2000). Compositional strategies of the hip-hop turntablist. Organised Sound, 5(2).

Wang, G. and Cook, P. (2003). ChucK: A concurrent, on-the-fly audio programming language. In Proceedings of the International Computer Music Conference, Singapore.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N. and Alexander, A. (2004). Live Algorithm Programming and a Temporary Organisation for its Promotion. Proceedings of the README software art conference, Aarhus, Denmark.

Wiedenbeck, S. (1986) Beacons in computer program comprehension. International Journal of Man-Machine Studies, 25, 697-709.