# Patterns of User Experience in Performance Programming

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

## ABSTRACT

This paper presents a pattern language for user experiences in live coding. It uses a recently defined analytic framework that has developed out of the Cognitive Dimensions of Notations and related approaches. The focus on performance programming offers particular value in its potential to construct rigorous accounts of the experiences of both performers (the live coder) and audiences. Although developed as an account of live coding, the findings promise to be relevant to a wider range of performance programming contexts, which could benefit from analysis in terms of live coding, if a systematic framework of this kind were available. The paper provides a detailed analytic commentary, drawing on the broadly diverse body of prior live coding practice, but using music live coding as a central running example. The findings demonstrate the advantage of rigorous analysis from an independent theoretical perspective, and suggest the potential for future work that might draw on this pattern language as a basis for empirical investigations of user experience, and as a theoretical grounding in practice-led design of new live coding languages and tools.

## 1. INTRODUCTION

The concept of *software patterns* – construction techniques that can be applied in many situations – is extremely well known in the software engineering industry, regularly applied by professionals and widely taught to undergraduates (e.g. Larman 2004, Holzner 2006). However, Blackwell and Fincher have argued (2010) that there has been a fundamental misunderstanding in the way this concept is being developed and applied. They observed that the software engineering researchers who originally coined the term had not intended it to be a catalogue of construction techniques, but rather based it on an analogy to the writing of architect and design philosopher Christopher Alexander (1978).

Alexander created the term *pattern language* to describe his systematized observations of the ways that people actually use and inhabit their built environment. He argued that buildings are shaped as much by their users as by materials and construction, and that it is useful for architects to have a systematic description of the things users experience. An example is the pattern "light on two sides of a room". This might be achieved, for example, when an architect, builder or home renovator installs glass windows in two different walls. However, the essence of the pattern is a description of the experience of light, not a prescription for achieving it (for example, by specifying how to place windows, what they are made out of, or even whether they are windows at all). Alexander's pattern language is a systematic collection of experience descriptions, not a set of rules for construction.

The goal of this paper is to apply an analogous approach to the study of the human context in which live coding is done, in order to achieve a systematic collection of descriptions of the user experiences that constitute live coding. Such a collection is potentially valuable both as a design resource (drawing attention to new opportunities for live coding tools or languages) and a critical resource (providing a vocabulary with which to describe the tools and languages of today).

The application of Alexander's pattern language to software was originally proposed by Kent Beck and Ward Cunningham (1987) and popularized in the form that is familiar today by the "Gang of Four": Gamma, Helm, Johnson and Vlissides (1994). The original analogy between software engineering and architecture was clearly expressed by Richard Gabriel as follows:

> *Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently. It should be clear that, in our context,* ***a "user" is a***

> ***programmer*** *who is called upon to maintain or modify software; a user is not (necessarily) the person who uses the software. In Alexander's terminology, a user is an inhabitant* (Gabriel 1993, emphasis added)

Subsequent extension of this work to user interface patterns (e.g. Tidwell 2005) has abandoned this insight from Beck, Cunningham and Gabriel that Alexander's ideas could be used to understand contexts where 'a "user" is a programmer' as in the quote above. In most user interface '"patterns", the user does *not* get to be a programmer, and does not have the creativity and control that arises from programming. Blackwell and Fincher (2010) argue that this situation may have arisen from misunderstanding of the original analogy, which has become more widespread as other "patterns" publications rely on secondary sources rather than the original publications by Beck and Cunningham. However, live coding offers an ideal opportunity to draw away from the (mis)understanding that may be implicit in the UI patterns community, and instead refocus on the creative power of programming, and on source code as the primary medium of interaction, rather than secondary to some other UI. Live coding is especially interesting because of the contrast it offers to mainstream software engineering (Blackwell and Collins 2005). By considering Alexander's (and Gabriel's) application of design patterns, the goal of this paper is to understand what it is that makes live coding distinctive, in relation to more conventional software engineering, and to other kinds of user experience.

The remainder of this paper is structured as follows. This introduction is fleshed out, first with a more precise characterization of performance programming, followed by more detail of Blackwell and Fincher's critique of conventional software and user interface "patterns", and then an introduction to their alternative pattern language as it applies to live coding. The main body of the paper describes that pattern language in detail, as the main contribution of the paper. A closing discussion reviews the validity and potential value of this approach.

### 1.1 The User Audience of Performance Programming

This paper takes a particular interest in the frequent interpretation of live coding as performance programming. Although live coding can be interpreted in other ways (in particular, as a software engineering method, as an educational strategy, or as a creative medium), the performance context introduces the key dynamic between the performer and audience. This dynamic immediately highlights many of the trade-offs that are inherent in the varieties of user experience captured by pattern languages. In particular, it is understood that performers and audiences have *different* experiences – a factor that is often neglected in the implicit universalism of software engineering methodology, and in mainstream theoretical research into principles of programming languages.

The performance context addressed by this paper can be understood primarily in terms that are familiar conventions of art-coding – where the code is written by a performer on stage, with a projection screen displaying the code as a stage backdrop, and the audience are either dancing (at an algorave), or sitting (in a concert/recital setting). This conventional layout might be varied slightly, for example when the performance accompanies a conference talk, when the live coder is providing background music at a function, or in a mixed genre variety show or cabaret context. However, the observations in the paper are also applicable to other presentation formats that maintain the performer/audience distinction, but replicated in digital media. In particular, video of live coding performances, whether recorded in one of the settings already described, or created specifically for online distribution, attracts larger audience numbers than concerts and algoraves.

The findings from this paper are also relevant to a further, perhaps even more widely viewed, online genre – the programming screencast. Screencasts are often recorded from a live 'performance', but typically focus on demonstrating software skills, or the features of software development tools, rather than on recording a stage performance. There is already substantial overlap between these genres and live coding as understood at this conference, for example in Sam Aaron's tutorial introduction *How to Hack Overtone with Emacs*[1]. Of course screencasts of art-coding are a minority interest by comparison to "performances" of programming with mainstream tools, whether tutorials such as *Data Driven Programming in Haskell*[2] or exhibitions of skill

[1] https://vimeo.com/25190186
[2] https://www.youtube.com/watch?v=045422s6xik

by popular coding celebrities such as Minecraft developer Notch[3]. The recent launch of a dedicated site (www.livecoding.tv) for performances of this kind suggests that this trend will continue to increase.

In the remainder of the paper, I assume for simplicity of presentation that the performer is creating music through live coding. I therefore use examples from music to illustrate the kinds of information structure involved in code. However, it is intended that these observations should apply equally to other live coding environments controlling other media – for example, my own Palimpsest system for live coding of visual imagery (Blackwell 2014), as well as the games, data analytics and software engineering examples above.

### 1.2 Patterns of User Experience

A key assumption of the presentation in this paper is that conventional uses of the term "pattern language" in software engineering and in human-computer interaction are based on a *misunderstanding* of Beck, Cunningham and Gabriel's original analogy between programming and architecture (Blackwell and Fincher 2010). This paper is therefore *not* primarily concerned with "software design patterns" as presented in Gamma et al (1994) and many other texts. Blackwell and Fincher (2010) argue that those texts are concerned mainly with construction techniques, rather than user experiences. This paper is particularly *not* concerned with "user interface patterns", as presented in many HCI texts and resource sites (e.g. http://ui-patterns.com/) which collect examples of specific construction techniques by analogy to software design patterns. As noted in the introduction to this paper, Blackwell and Fincher (2010) argue that this literature continues the misunderstanding of pattern languages, through a focus on construction of user interfaces rather than the user experience of programming. Although it has been necessary to cite this literature in order to avoid further misunderstanding, this paper will not make any further reference to either software patterns or user interface patterns literatures.

Blackwell and Fincher's proposal for the creation of a pattern language of user experience in software was modeled closely on the Cognitive Dimensions of Notations (CDs) framework originally proposed by Thomas Green (1989), greatly expanded by Green and Petre (1996), and incrementally developed with many collaborators since then, including the present author and many others. Several alternative perspectives on CDs have been proposed, introducing corresponding approaches to tangible user interfaces (Edge 2006), collaborative representations (Bresciani 2008) and several others. Blackwell and Fincher argued that the goal of CDs was far more closely related to Alexander's pattern language, in that it aimed to provide a vocabulary with which the designers of programming languages and other complex notations could discuss the usability properties of those notations. An example CD was "viscosity" – the experience of using a system in which small changes are unreasonably hard to achieve, like "wading through treacle". The vocabulary was grounded in previous literature, but was a "broad brush" description rather than focusing on details of the kind that can be measured in controlled laboratory studies, or time-and-motion style exploration of efficiency in use. The popularity of the framework arose in part because labels such as "viscosity" captured intuitive feelings that were recognized by many programmers as aspects of their everyday experience, but aspects that they had not previously had a name for.

In the 25 years since Green originally proposed the Cognitive Dimensions of Notations framework, it has been highly influential as an approach to understanding the usability of programming languages. The paper by Green and Petre (1996) was for many years the most highly cited original paper to be published in the Journal of Visual Languages and Computing, and a 10th anniversary issue of that journal reviewed the many ways in which the framework had been applied, including its application in mainstream commercial programming products such as Microsoft's Visual Studio (Dagit et al. 2006, Green et al. 2006). This paper cannot review that literature in detail, but it extends to practical design guidance for programming languages and APIs, evaluation methods for programming tools, theories of user interaction, requirements capture methods and others.

Based on this substantial and diverse body of research, the author has developed a teaching guide, which synthesizes a pattern language of user experience from all these different sources, variants and extensions of the original CDs framework. This guide has been used in contexts that are intentionally distinct from the programming language research that has been the main focus of CDs until now. The most substantial application to date has been in a graduate course within a professional master's programme for sustainability leadership[4]. The goal of that course has been to highlight the role played by representation systems, when global challenges are being negotiated in the presence of technology. It is taught to students

[3] https://www.youtube.com/watch?v=rhN35bGvM8c

with no prior experience of interaction design, using a case study of mobile GPS usage by indigenous people negotiating with logging companies in the Congo Basin (Lewis 2014). This course has provided an opportunity to expand the concept of a representational pattern language to the widest possible extent, encompassing considerations of physical context, language, culture, colonialism, embodiment, ecosystems, distributed and locative media, and many others.

### 1.3 Interrogating User Experience in Performance Programming

Starting from a pattern language that has been independently developed to analyse the user experience of novel digital interventions, this paper now analyses performance programming from that perspective. The goal of the analysis is to explore the ways in which performance programming exhibits (or does not exhibit) patterns that are also seen in other digital media contexts. These patterns are expressed, as in the Cognitive Dimensions of Notations framework, in terms of interaction with an *information structure*, where the structure is (in the digital music context) the configuration of synthesisers, samples, filters, instruments and other software components that generate sound through digital means.

In keeping with mainstream practice of analysis using CDs, the patterns are used as a discussion tool, providing a structured basis for systematic consideration of different aspect of user experience. In classical CDs analysis, the characteristic user activities are first defined, together with a profile of dimensions that are most salient for that kind of activity. Each dimension is then considered in turn, in order to observe how the particular combination of notation and environment will support the identified activities. This consideration also includes trade-offs where improvement on one dimension might result in detriment for another, and possible work-arounds that users would engage in. Each of these elements has been included in the following analysis, using appropriate analogies to the broader class of user experience patterns now identified.

The specific form in which the patterns are expressed below, using reference numbers to identify particular patterns, allows the reader to cross-reference this analysis with a forthcoming publication in which the patterns of user experience have been applied in another non-programming context, to the design of novel diagrammatic representations (Blackwell in press). An appendix in that publication can also be consulted for more detailed derivations that relate each pattern to the previous bodies of research from which is has been developed, in particular drawing on the various iterations and variants of the Cognitive Dimensions of Notations framework.

## 2. PATTERNS OF ACTIVITY IN PERFORMANCE PROGRAMMING

As with the architectural analyses of Christopher Alexander, patterns in the user experience of live coding can be described from different perspectives. This section contrasts the different modes in which people interact with code, expressed as types of activity that are characteristic of *Interpretation*, *Construction* and *Collaboration* activities. In each case, these are activities that have been observed in other areas of notation and representation use, and this pre-existing descriptive framework is used to interrogate different aspects of performance programming, stepping aside from conventional descriptions that are already familiar in the critical discourse of live coding.

Each of these kinds of activity, as with the more limited set of activities that has previously been described in the Cognitive Dimensions framework, is associated with a different profile of user experience patterns that are found to be particularly salient in the context of that activity. These profiles are briefly indicated, in the following discussion, using reference numbers referring to the experience patterns that will be described later in the paper. This numbering scheme is consistent with (Blackwell in press), to allow further citations of previous work.

### 2.1 Interpretation activities: reading information structure

In the performance programming setting, the audience interprets the information structure that the performer constructs. Interpretation activities include: **Search**: The audience often follow the current cursor position, but may also search for the code that was responsible for producing a particular audible effect. This activity is enabled by patterns VE1, VE4, SE3, TE4 below. **Comparison**: The audience constantly compares different pieces of code – both diachronically (the state of the code before and after a change), and

[4] Master of Studies at the Cambridge Institute for Sustainability Leadership
http://www.cisl.cam.ac.uk/graduate-study/master-of-studies-in-sustainability-leadership

synchronically (comparing expressions on different parts of the screen in order to infer language features by comparison). This activity is enabled by patterns VE5, SE4, ME4, TE2 below. **Sense-making**: Live coding audiences often have no prior expectation of the code that is about to be produced, so they must integrate what they see into a mental model of its overall structure. This activity is enabled by patterns VE2, VE3, SE1, ME1, ME3, TE3, TE5 below.

### 2.2 Construction activities: building information structure

The performer is defining, in various layers, an abstract syntax tree, a synthesiser configuration, or a musical structure with audible consequences that will be perceived by listeners. The following activities represent different approaches to the creation of this structure. **Incrementation:** The structure of the code has been established, but a piece of information is added – e.g. another note, a synthesiser parameter. This activity is enabled by patterns IE1, PE6 below. **Transcription:** The structure is already defined, and the performer needs to express it in the form of code. Perhaps it has been rehearsed or pre-planned? This activity is enabled by patterns ME2, IE2, IE3, IE5, PE2, PE5 below. **Modification:** Is a constant of live coding – many live coders commence a performance with some code that they may modify, while even those who start with a blank screen restructure the code they have written. This activity is enabled by patterns SE2, ME5, IE4, TE1, PE1, CE1 below. **Exploratory design:** Despite the musically exploratory nature of live coding, it is rare for complex data structures or algorithms to be explored in performance – primarily because an executing program can only gradually change its structure while executing. As a result, live coding is surprisingly unlikely to exhibit the kinds of "hacking" behaviour anticipated when the distinctive requirements of exploratory software design were highlighted by the Cognitive Dimensions framework. This activity is enabled by patterns TE5, PE3, PE4, CE2, CE3, CE4 below.

### 2.3 Collaboration activities: sharing information structure

Descriptions of collaboration were added relatively recently to the Cognitive Dimensions framework (Bresciani et al 2008), perhaps because the stereotypical expectation is that programming is a relatively solitary activity. However, the performance programming context immediately introduces a social and "collaborative" element, if only between the performer and audience. The activities that have been identified in previous work include: **Illustrate a story**: Originally intended to describe situations in which a notation is used to support some other primary narrative, this provides an interesting opportunity to consider the audience (and performer) perspective in which the music is the primary medium, and the code, to the extent it is shared, is there to support this primary function. This activity is enabled by patterns VE2, VE4, IE6, TE1, CE3 below. **Organise a discussion:** At first sight, this seems a foreign perspective to live coding. An algorave is not a discussion. But one might consider performance settings such as conferences, or comments on online video, in which the audience do respond to the work. Does the code play a part? Furthermore, on-stage improvisation among a group of performers can be analysed as discursive – does the code support this? This activity is enabled by patterns ME5, IE2, TE2, PE3, PE4, CE4 below. **Persuade an audience**: What is the visual rhetoric embedded in code? Many notation designers are reluctant to admit the rhetorical or connotative elements of the representations they create, and these design elements of live coding languages are often left relatively tacit. This activity is enabled by patterns VE3, SE4, ME2, ME6, IE5, TE3, TE5 below.

## 3. DESIGN PATTERNS FOR EXPERIENCE IN USE

This section offers a comprehensive description of patterns in user experience that result from, or are influenced by, the design of different notations or tools. As discussed in the previous section, different aspects of the performance programming context are enabled by different subsets of these patterns. This set of patterns can potentially be used either as a checklist of design heuristics, for those who are creating new live coding tools, or as a means of reflecting on the capabilities of the tools they already have. In particular, there are many design decisions that carry *different* implications for audiences and performers, which would appear to be an important critical consideration for the design and evaluation of performance programming technologies.

### 3.1 Experiences of Visibility

Visibility is essential to performance programming – the TOPLAP manifesto demands "show us your screens."[5] It is therefore completely appropriate that this group of patterns so often comes before all others. **VE1: The information you need is visible:** The constraints imposed by the performance programming context are extreme – a single screen of code is projected or streamed, in a font that must be large enough for an audience to see. **VE2: The overall story is clear:** Can code be presented in a way that the structure is apparent, or might it be necessary to leave out some of the detail in order to improve this overall understanding? **VE3: Important parts draw your attention:** When there is such a small amount of code visible, perhaps this isn't much of a problem. However, it seems that many live coders support their audiences by ensuring that the current insertion point is prominent, to help in the moment engagement with the performance. **VE4: The visual layout is concise:** At present, the constraints already described mean that the importance of this pattern is unavoidable. **VE5: You can see detail in context:** This seems to be an interesting research opportunity for live coding. At present, many live coding environments have adopted the buffer-based conventions of plain text editors, but there are better ways of showing structural context, that could be borrowed from other user interfaces, IDEs or screen media conventions.

### 3.2 Experiences of Structure

In the running example used in this paper, it is assumed that the "structures" implicit in the performance programming context are the structures of music synthesis. However, as already noted, this running example should also be interpreted as potentially applying to other kinds of live-coded structured product – including imagery, dance and others. In all these cases, the understanding inherited from the Cognitive Dimensions framework is that the information structure consists of 'parts', and relationships between those parts. **SE1: You can see relationships between parts:** Are elements related by a bounded region, or lines drawn between them? Do multiple marks have the same colour, orientation, or size? Are there many verbal or numeric labels, some of which happen to be the same, so that the reader must remember, scan and compare? **SE2: You can change your mind easily:** This is a critical element supporting the central concern with improvisation in live coding. The problems associated with this pattern were captured in Green's most popular cognitive dimension of "Viscosity – a sticky problem for HCI" (1990). **SE3: There are routes from a thing you know to something you don't:** Virtuoso performers are expected to hold everything in their heads. Would it be acceptable for a live coder to openly rely on autocompletion, dependency browsers, or API tutorials when on stage? **SE4: You can compare or contrast different parts:** The convention of highly constrained screen space means that juxtaposition is usually straightforward, because everything necessary is already visible, allowing the performer to judge structural relationships and the audience to interpret them.

### 3.3 Experiences of Meaning

In many programming languages, the semantics of the notation are problematic, because the programmer must anticipate the future interpretation of any given element. In live coding, meaning is directly available in the moment, through the concurrent execution of the program being edited. As a result, there is far greater freedom for live coding languages to explore alternative, non-literal, or esoteric correspondences. **ME1: It looks like what it describes:** Live coders often use this freedom for a playful critique on conventional programming, with intentionally meaningless variable names, ambiguous syntax and so on. **ME2: The purpose of each part is clear:** Within the granularity of the edit/execute cycle, the changes that the audience perceive in musical output are closely coupled to the changes that they have observed in the source code. As a result, "purpose" is (at least apparently) readily available. **ME3: Similar things look similar**: At present, it is unusual for live coders to mislead their audiences intentionally, by expressing the same behaviour in different ways. This could potentially change in future. **ME4: You can tell the difference between things:** In contrast, live coding languages often include syntax elements that may be difficult for the audience to distinguish. A potential reason for this is that the resulting ambiguity supports a richer interpretive experience for the audience. This is explored further in CE2 and CE3 below. **ME5: You can add comments:** A simplified statement of the cognitive dimension of secondary notation, this refers to those aspects of the code that do not result in any change to the music. In performance, the text on the screen offers a side-band of communication with the audience, available for commentary, meta-narratives and many other intertextual devices. **ME6: The visual connotations are appropriate:** The visual aesthetic

[5] http://toplap.org/wiki/ManifestoDraft

of live coding environments is a highly salient aspect of their design, as appropriate to their function as a performance stage set. The muted greys of Marc Downie's *Field*, the ambient radar of Magnusson's *Threnoscope*, the vivid pink of Aaron's *Sonic Pi*, and the flat thresholded geometry of my own *Palimpsest* are all distinctive and instantly recognisable.

### 3.4 Experiences of Interaction

These patterns relate to the user interface of the editors and tools. It is the performer who interacts with the system, not the audience, so these are the respects in which the greatest tensions might be expected between the needs of the two. **IE1: Interaction opportunities are evident:** This is a fundamental of usability in conventional user interfaces, but the performance of virtuosity involves the performer giving the impression that all her actions are ready to hand, perhaps in a struggle against the obstacles presented by the "instrument". Furthermore, the performer may not want the audience to know what is going to happen next – for example, in my own performances with Palimpsest, I prefer the source images that I intend to manipulate to be arranged off-screen, on a second monitor that is not visible to the audience, so that I can introduce them at the most appropriate points in the performance. The question of choice is always implicit, but a degree of mystery can enhance anticipation. **IE2: Actions are fluid, not awkward:** Flow is essential to both the experience and the illusion of performance. For many developers of live coding systems, this is therefore one of their highest priorities. **IE3: Things stay where you put them:** Predictability is a virtue in most professional software development contexts, but in improvised performance, serendipity is also appreciated. As a result, the representations are less static than might be expected. An obvious example is the *shift* and *shake* features in Magnusson's ixi lang, whose purpose is to behave in opposition to this principle. **IE4: Accidental mistakes are unlikely:** On the contrary, accidents are an opportunity for serendipity, as has been noted since the earliest discussions of usability in live coding (e.g. Blackwell & Collins 2005). **IE5: Easier actions steer what you do:** In many improvising genres, it is understood that the affordances of the instrument, of the space and materials, or of the body, shape the performance. However, easy licks quickly become facile and trite, with little capacity to surprise the audience. As a result, every new live coding tool can become the starting point for a miniature genre, surprising at first, but then familiar in its likely uses. **IE6: It is easy to refer to specific parts:** Critical commentary, education, support for user communities, and the collaborative discourse of rehearsal and staging all require live coding performers to talk about their code. Although ease of reference is seldom a salient feature early in the lifecycle of a new tool, it becomes more important as the audience expands and diversifies.

### 3.5 Experiences of Thinking

The kinds of thinking done by the performer (creative decisions, planning the performance, analysing musical structures) are very different to those done by the audience (interpretation, active reception, perhaps selecting dance moves). Both appear rather different to conventional professional programming situations, which far more often involve reasoning about requirements and ways to satisfy them. **TE1: You don't need to think too hard**: Cognitive load, resulting from working memory and dependency analysis, is less likely to be immediately challenging in live coding situations because both the code and the product are directly available. However, audiences often need to interpret both the novel syntax of the language, and the performer's intentions, meaning that they may (unusually) have greater cognitive load than the performer – if they are concentrating! **TE2: You can read-off new information**: Purely textual languages are unlikely to exhibit this property, but visualisations such as Magnusson's *Threnoscope* exhibit relations between the parts beyond those that are explicit in the code, for example in phase relations that emerge between voices moving with different velocity. **TE3: It makes you stop and think**: The challenge noted for audiences in TE1 here becomes an advantage, because it offers opportunities for richer interpretation. **TE4: Elements mean only one thing**: Lack of specificity increases expressive power at the expense of abstraction, often resulting in short-term gains in usability. In live coding, the performer is usually practiced in taking advantage of the available abstraction, while the audience may appreciate interpretive breadth. **TE5: You are drawn in to play around:** Clearly an advantage for performing programmers, and less problematic in a context where risk of the program crashing is more exciting than dangerous. However, playfulness is supported by the ability to return to previous transient states (IE3).

### 3.6 Experiences of Process

Many conventional software development processes include assumptions about the order in which tasks should be done – and this is less common in live coding. However, it is useful to remember that rehearsal and composition/arrangement do involve some different processes that may not be involved in performance situations (for example, Sam Aaron notes in another paper at this conference that he keeps a written diary while rehearsing). **PE1: The order of tasks is natural:** Every user may have a different view of what is 'natural', although live coding audiences appreciate an early start to the music. **PE2: The steps you take match your goals:** Live coding is a field that shies away from prescription, and 'goals' often emerge in performance. Nevertheless, IE5 results in recognisable patterns that may not have been conceived explicitly by the performer. **PE3: You can try out a partial product:** Absolutely inherent in live coding – a 'finished' live coded program seems almost oxymoronic! **PE4: You can be non-committal:** Another high priority for live-coded languages, which often support sketch-like preliminary definitions. **PE5: Repetition can be automated:** Every performing programmer is aware of developing personal idiomatic conventions. Occasionally the language is extended to accommodate these automatically, but more often, they become practiced in the manner of an instrumental riff, or encoded through muscle memory as a 'finger macro.' **PE6: The content can be preserved:** Some live coders habitually develop complex libraries as a jumping-off point for improvisation (e.g. Andrew Sorensen), while others prefer to start with an empty screen (e.g. Alex McLean). Whether or not code is carried forward from one performance to another, it might be useful to explore more thoroughly whether the temporality of the performance itself can be preserved, rather than simply the end result.

### 3.7 Experiences of Creativity

In analysis of conventional software engineering, this final class of experience might appear trivial or frivolous. However, when programming is taking place in a performing arts context, 'creativity' (of some kind) is assumed to be one of the objectives. This suggests that these patterns will be among the most valued. **CE1: You can extend the language:** Live coders constantly tweak their languages, to support new performance capabilities, but this most often happens offline, rather than during performance. Language extension in performance is conceivable, and may offer interesting future opportunities, but would greatly increase the challenge to audience and performer in terms of TE4. **CE2: You can redefine how it is interpreted:** The predefined semantics of most mainstream programming languages means that open interpretation may be impractical. But in live coding, opportunities for increased expressivity, and serendipitous reinterpretation, suggest that this experience is likely to be more highly valued. **CE3: You can see different things when you look again:** This is a key resource for inspiration in improvised performance contexts, to generate new ideas for exploration, and to provide a wider range of interpretive opportunities for the audience. As a result, this represents one of the most significant ways in which live coding languages differ from conventional programming languages, where ambiguity is highly undesirable. **CE4: Anything not forbidden is allowed:** Many programming language standards attempt to lock down the semantics of the language, so that all possible uses can be anticipated and tested. In creative contexts, the opposite is true. An execution result that is undesirable in one performance may be completely appropriate in another, and digital media artists have often revelled in the glitch or crash, as a demonstration of fallibility or fragility in the machine.

## 4. DISCUSSION

This paper has investigated the nature of user experience in live coding, and also in other contexts that might more broadly be described as performance programming, characterized by the presence of an audience observing the programming activity. This investigation has proceeded by taking an existing analytic framework that describes patterns of user experience, in the style of an architectural pattern language, and has explored the insights that can be obtained by considering performance programming from the perspective of this framework.

### 4.1 Validity of the Analysis

The validity of this analysis relies on two precautions: Firstly, the existing framework is applied without modification or adaptation. Rather than selecting elements from the framework by convenience or apparent surface relevance, the structure and content of the existing framework has been applied without modification. This provides an opportunity for disciplined consideration of all possible factors, whether or

not they might have been considered to be relevant at first sight. This first precaution has demonstrated some novel advantages, for example by drawing closer attention to the user experience of the audience in performance programming situations. The value of this precaution can be understood by contrast with previously less successful attempts to apply the Cognitive Dimensions framework, in which researchers ignored, or redefined, particular dimensions that were inconvenient or inconsistent with their assumptions (as discussed in Blackwell & Green 2000).

The second precaution is that the framework chosen for application is intentionally tangential to the existing analytic approaches that have been applied in live coding, thus offering a form of qualitative triangulation. There have been previous analyses of live coding from the perspective of performance studies, and of programming language design. Although these prior analyses have resulted in a useful emerging body of theory, there is a tendency for such bodies of theory to be self-reinforcing, in that analytic design discourse is grounded in the body of theory, and empirical investigations of its application reflect the same assumptions. In contrast, the framework applied here was developed for analysis of visual representation use, in situations ranging from central African forests to Western business, education and domestic settings. Although drawing heavily on the approach pioneered in the Cognitive Dimensions of Notations, this framework has subsequently been adapted for an audience of visual representation designers, with many changes of terminology and structure (Blackwell in press). One consequence is that the terminology may appear less immediately relevant to live coding than is the case with familiar Cognitive Dimensions such as secondary notation. But the consequent advantage is to defamiliarise the discussion of user experience in live coding, allowing us to achieve a systematic analysis from a novel perspective, and an effective triangulation on the understanding of interaction in live coding.

### 4.2 Applying the Analysis

The next phase of this work is to apply the results of this analytic investigation as a starting point for empirical study and practice-led design research. The immediate opportunity for empirical study is to use the findings from this investigation as a coding frame[6] for the analysis of qualitative descriptions of user experiences in performance programming, whether taken from audiences or from performers. The framework will be beneficial to such situations, by providing a theoretically-grounded organisational structure rather than an *ab initio* categorisation. There is also a possible advantage for the framework itself, in that new sources of experience reports, from situations as different as algoraves and screencast videos, may identify new patterns of experience that have not previously been noticed in other programming and formal representation contexts.

The second opportunity is to use the results of this analysis as a resource for design research. This pattern language of user experience in performance programming retains the potential of the original Cognitive Dimensions framework, that it can be employed as a discussion vocabulary for use by language designers. It points to design opportunities, as noted at points throughout the above analysis. It offers the potential for reflection on the consequences of design decisions. It draws attention to known trade-offs, although this paper does not have space for the detailed listing of those trade-offs described elsewhere (Blackwell in press). Finally, it offers the potential for use as an evaluative framework, for example in the manner of Heuristic Evaluation, or as a basis for the design of user questionnaires, in the manner of the Cognitive Dimensions questionnaire (Blackwell & Green 2000).

The discussion of live coding in this paper has primarily drawn on music live coding systems to illustrate the relationship between structure and representation. This choice was made in part because the author is more familiar with music than with other performing arts. However, it is intended that this analysis can be applied to many other arts contexts, and the author's preliminary investigations have already included study of dance (Church et al. 2012) and sculpture (Gernand et al. 2011), as well as experiments in live-coded visual imagery (Blackwell & Aaron 2014). Current live coding research by others, for example McLean's performances of textile craft practices such as knitting and weaving, is also likely to benefit from this analytic approach.

## 5. CONCLUSIONS

Systematic description of user experience is a valuable design resource, as noted in the development and subsequent popularity of Christopher Alexander's pattern language for the design of the built environment.

[6] Note that the term "coding frame" is used here as a technical term in qualitative data analysis, which has no relationship to "coding" of software.

The idea of the pattern language has been appropriated enthusiastically in software engineering and in user interface design, although in a manner that has lost some of the most interesting implications in studying experiences of computation. Live coding, and performance programming more broadly, represent distinctive kinds of user experience. The user experiences of performance programming can be analysed in terms of the distinct ways that performers and audiences use and interpret the representations that they see. As with Cognitive Dimensions of Notations, different types of experience result from the ways that information structures are represented and manipulated. In live coding contexts, these information structures must be mapped to the structural elements by which art works are constructed and interpreted. The resulting framework offers a basis for the analysis of live coding experiences, as well as for reflexive critical understanding of the new languages and tools that we create to support live coding.

**Acknowledgments**

Thank you to Thomas Green and Sally Fincher, for valuable review and contributions to the development of the Patterns of User Experience framework.

## REFERENCES

Alexander, C. (1978). *The timeless way of building*. Oxford University Press.

Beck, K. and Cunningham, W. (1987). *Using pattern languages for object-oriented programs.* Tektronix, Inc. Technical Report No. CR-87-43, presented at OOPSLA-87 workshop on Specification and Design for Object-Oriented Programming. Available online at http://c2.com/doc/oopsla87.html (accessed 17 September 2009)

Blackwell, A.F. (2014). Palimpsest: A layered language for exploratory image processing. *Journal of Visual Languages and Computing* **25**(5), pp. 545-571.

Blackwell, A.F. (in press) A pattern language for the design of diagrams. In C. Richards (Ed), *Elements of Diagramming*. To be published June 2015 by Gower Publishing.

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of the Psychology of the Programming Interest Group (PPIG 2005),* pp. 120-130.

Blackwell, A.F. & Fincher, S. (2010). PUX: Patterns of User Experience. *interactions* 17(2), 27-31.

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. and Aaron, S. (2014). Take a little walk to the edge of town: A live-coded audiovisual mashup. Performance/presentation at CRASSH Conference Creativity, Circulation and Copyright: Sonic and Visual Media in the Digital Age. Centre for Research in the Arts, Social Sciences and Humanities, Cambridge, 28 March 2014.

Bresciani, S., Blackwell, A.F. and Eppler, M. (2008). A Collaborative Dimensions Framework: Understanding the mediating role of conceptual visualizations in collaborative knowledge work. *Proc. 41st Hawaii International Conference on System Sciences (HICCS 08),* pp. 180-189.

Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2012),* pp. 163-174.

Dagit, J., Lawrance, J., Neumann, C., Burnett, M. Metoyer, R. and Adams, S. Using cognitive dimensions: Advice from the trenches. *Journal of Visual Languages & Computing*, 17(4), 302-327.

Edge, D. and Blackwell, A.F. (2006). Correlates of the cognitive dimensions for tangible user interface. *Journal of Visual Languages and Computing*, 17(4), 366-394.

Gabriel, R.P. (1993). Habitability and piecemeal growth. *Journal of Object-Oriented Programming* (February 1993), pp. 9-14. Also published as Chapter 2 of *Patterns of Software: Tales from the Software Community*. Oxford University Press 1996.
Available online http://www.dreamsongs.com/Files/PatternsOfSoftware.pdf

Gamma, E. Helm, R. Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gernand, B., Blackwell, A. and MacLeod, N. (2011). *Coded Chimera: Exploring relationships between sculptural form making and biological morphogenesis through computer modelling*. Crucible Network.

Green, T.R.G. (1989). Cognitive Dimensions of Notations. In L. M. E. A. Sutcliffe (Ed.), *People and Computers V.* Cambridge: Cambridge University Press.

Green, T.R.G. (1990). The cognitive dimension of viscosity: A sticky problem for HCI. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction (INTERACT '90),* pp. 79-86.

Green, T.R.G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 131-174.

Green, T.R.G., Blandford, A.E., Church, L., Roast, C.R., and Clarke, S. (2006). Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4), 328-365.

Holzner, S. (2006). *Design Patterns For Dummies*. Wiley.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall.

Lewis, J. (2014). Making the invisible visible: designing technology for nonliterate hunter-gatherers. In J. Leach and L. Wilson (Eds). *Subversion, Conversion, Development: Cross-Cultural Knowledge Exchange and the Politics of Design*, 127-152.

Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly.