

Collaboration and learning through live coding

Edited by

Alan Blackwell¹, Alex McLean², James Noble³, and
Julian Rohrhuber⁴

1 University of Cambridge, GB, alan.blackwell@cl.cam.ac.uk

2 University of Leeds, GB, a.mclean@leeds.ac.uk

3 Victoria University of Wellington, NZ, k.jx@ecs.vuw.ac.nz

4 Robert Schumann Hochschule für Musik – Düsseldorf, DE,
julian.rohrhuber@musikundmedien.net

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13382 “Collaboration and learning through live coding”. Live coding is improvised interactive programming, typically to create electronic music and other digital media, done live with an audience. Our seminar was motivated by the phenomenon and experience of live coding. Our conviction was that those represent an important and broad, but seldom articulated, set of opportunities for computer science and the arts and humanities. The seminar participants included a broad range of scholars, researchers, and practitioners spanning fields from music theory to software engineering. We held live coding performances, and facilitated discussions on three main perspectives, the humanities, computing education, and software engineering. The main outcome of our seminar was better understanding of the potential of live coding for informing cross-disciplinary scholarship and practice, connecting the arts, cultural studies, and computing.

Seminar 15.–20. September, 2013 – www.dagstuhl.de/13382

1998 ACM Subject Classification J.5 [Computer Applications]: Arts and humanities—Performing arts, K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education, D.2.1 [Software Engineering]: Requirements/Specifications—Methodologies

Keywords and phrases Live coding, Collaboration, Learning, Improvised interactive programming, Computer music, Algorithmic composition, TOPLAP

Digital Object Identifier 10.4230/DagRep.3.9.130

Edited in cooperation with Jochen Arne Otto (ZKM | Center for Art and Media Karlsruhe, DE)

1 Executive Summary

Robert Biddle

Alex McLean

Alan Blackwell

Julian Rohrhuber

License  Creative Commons BY 3.0 Unported license
© Robert Biddle, Alex McLean, Alan Blackwell, and Julian Rohrhuber

The goal of this seminar was to understand and develop the emerging practice, characteristics and opportunities in live coding, with an emphasis on three perspectives: the humanities, computing education, and software engineering. The opening days of the seminar were broadly structured to provide thematic introductions followed by facilitated discussions on



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Collaboration and learning through live coding, *Dagstuhl Reports*, Vol. 3, Issue 9, pp. 130–168

Editors: Alan Blackwell, Alex McLean, James Noble, and Julian Rohrhuber



DAGSTUHL
REPORTS

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

each of these three perspectives. These were interspersed with live coding performances and experiments, in order to ensure that theoretical concerns remained grounded within this discipline that fundamentally blurs the separation of concerns between theory and practice.

The second half of the seminar was problem-oriented, resulting in concrete progress on specific technical topics, together with development of a research roadmap, publications and policy strategy to realise the significant benefits that live coding promises in a number of fields. Finally, in the spirit of both practice as a form of theory and theory as a form of practice, the seminar included some exciting musical experiences – an Algorave club night in London, with performances by delegates who were traveling from other countries on their way to the seminar; an inter-continental collaborative performance hosted jointly with the IEEE VL/HCC conference in San Jose; a conceptual proposal for an interactive sound installation in the Schloss Dagstuhl garden; and live-coded jam sessions in venues ranging from the woods of the old castle, to evening cabaret entertainment in the beautiful Dagstuhl music room.

Our main findings in relation to the three contrasting research perspectives were as follows:

1. Live coding illuminates the ways in which programming can be an artistic practice, software-as-art, going beyond a mere supporting role, and illustrating that software is itself a cultural artefact, not simply an infrastructure commodity. We see many opportunities for nuanced, cross-disciplinary contributions to the digital humanities, for example in a revitalisation of the historical connection between computation and weaving, insights into the role of practice and experiment, and an enrichment of the notion of computation itself. Indeed, as computing becomes embedded in culture, the live, everyday authorship of computation becomes a socio-political question of freedom of speech and empowerment.
2. Live coding can play an important role in computing education, because it allows programming to be demonstrated and learned in a simple but authentic context. At the same time, it can support an *affective* teaching strategy where learners are not only motivated by the production of sound, visuals and other phenomena, but are also clear on the distinctly human activity which produces them. Thereby, however, it maintains a sense of discovery of something unanticipated and not prefigured. Of particular importance for learning is the potential for deeper engagement with the non trivial nature of computing, rather than an occupation with the operation of end-user application software.
3. Live coding offers new insights with regard to software engineering processes. The history of software engineering process can be seen as a move from heavyweight lock-step approaches to more agile approaches with fast cycles of development and feedback. At their heart, the new approaches rely on collaboration, as developers, designers, and customers work together to steer the process toward mutual success. Live coding demonstrates this kind of approach in a compelling way, with simple tools, a short time frame, but still allowing improvisational collaboration between performers and various audiences.

Perhaps more significant than any of these individual considerations is an ambitious holistic vision: that live coding can entirely change the way we think about programming. Indeed, the common experience articulated at the workshop is that live coding exemplifies both the power and the excitement of programming – in a small space, in a short time, available and accessible to anyone. Live coding exposes the *soul* of programming.

Our next steps are a series of collaborative workshops and programs to articulate and demonstrate this collection vision of a broad and expanding role for live coding.

This report is a collection of texts that were direct outcomes of the seminar rather than having been set up in advance. Taking the role of editor seriously, Jochen Arne Otto has

diligently collected, arranged and revised the heterogeneous materials, a synthesis without which such a concise paper would not have been possible. While taking responsibility for all remaining mistakes and omissions, we would like to thank Jochen very much for his commitment to this difficult task.

2 Table of Contents

Executive Summary

Robert Biddle, Alex McLean, Alan Blackwell, and Julian Rohrhuber 130

Overview of Talks

Live Coding as a Model for Cultural Practice
Geoff Cox 135

Live Coding in Education
Mark Guzdial 135

Live Coding, Software Engineering and Collaboration
Robert Biddle 136

Group Discussions

Critical Engineering and Software Tools
Roly Perera 138

Algorithmic Complementarity, or the Impossibility of “Live” Coding
Julian Rohrhuber 140

Tidal: Representing Time with Pure Functions
Alex McLean 142

Explicit vs. Implicit and External vs. Internal (Representations of) Time
Julian Rohrhuber 144

Stress and Cognitive Load
Alex McLean 145

Practical Implementation Session

Setting Up a Shared Temporal Frame
David Ogborn 146

Live Coding: Some History and Implementations

Personal Accounts on the History of Live Coding
Renate Wieser and Julian Rohrhuber 147

Extempore
Andrew Sorensen 148

Fluxus
Dave Griffiths 149

ixi lang
Thor Magnusson 150

SuperCollider and the Just In Time Programming Library
Julian Rohrhuber 150

Overtone
Samuel Aaron 152

Republic: Collaborative Live Coding 2003–2013
Alberto de Campo 152

Sonic Pi	
<i>Samuel Aaron</i>	153
Personal Reflections	
Clickety-Click: Live Coding and Software Engineering	
<i>Robert Biddle</i>	154
Cultural-Epistemological Aspects of Live Coding	
<i>Geoff Cox</i>	159
Dagstuhl: Live Coding and Memorable Discussions	
<i>Dave Griffiths</i>	160
Live Coding, Computer Science, and Education	
<i>Mark Guzdial</i>	162
Algorithms, Live Patterning and the Digital Craft of Weaving	
<i>Ellen Harlizius-Klück</i>	165
A Few Notes from the Seminar	
<i>Julian Rohrhuber</i>	166
Participants	168

3 Overview of Talks

3.1 Live Coding as a Model for Cultural Practice

Geoff Cox (Aarhus University, DK)

License  Creative Commons BY 3.0 Unported license
© Geoff Cox

Could a precedent for live coding be seen in Wolfgang von Kempelen’s 18th century chess-playing automaton, a description of which introduces Walter Benjamin’s essay “On the Concept of History”? The machine’s trickery demonstrates that the dynamic of history and that of the machine is fake; and the task of the historical materialist is to reveal the inner workings as ideological constructions.

This relates somewhat to my interest in live coding: how it seems prototypical of contemporary labour dynamics and what is hidden from view – inasmuch as labour has become more performative and linguistic in character (and precarious, with reference to Amazon’s Mechanical Turk – a crowdsourced online workplace – that exhibits so-called “artificial artificial intelligence”). Live coding makes apparent coding practice and embodiment (labour) and the visibility of code, coding and coders (“show us your screens”, they proclaim).

Expressed ever so briefly, my point is that in a culture where behaviour is increasingly prescribed by various scripts, scores, programs (and these are enclosed), live coding offers a way to understand these dynamics, and accordingly suggests ways out of various automatisms and determinisms associated with code.

3.2 Live Coding in Education

Mark Guzdial (Georgia Institute of Technology – Atlanta, US)

License  Creative Commons BY 3.0 Unported license
© Mark Guzdial

In one of the first talks at the Dagstuhl Seminar on Live Coding, Sam Aaron asked, “Why aren’t kids programming?” It’s fun, it’s creative, and it’s an inexpensive activity that’s easily accessible. Why isn’t it happening?

The problem is one of perception and access. As Simon Peyton-Jones of *Computing At School* has pointed out, most people define “Computer Science” as “a niche university subject for socially-challenged males,” when most computer scientists see it as “a foundational discipline, like maths or physics, that every child should learn.” In the United States, computer science is one of the least-subscribed topics in science, technology, engineering, and mathematics (STEM) in secondary schools. For the 30,000 high schools in the United States, there are only some 2,000 high school computer science teachers. There is very little access. Even if more students *wanted* to study computer science, it’s not available.

Computer science in the United States is predominantly male and white or Asian. The United States is 13.1% Black and 16.9% Hispanic. In most of the states (37 out of 50), fewer than 14 Black students per state took the Advanced Placement Exam in Computer Science (similar to an A-Level) – not percentage, 14 *individuals*. California, the largest and most Hispanic state in the US (38% of population), has a rate of Hispanic students taking the AP CS exam of less than 10%.

How do we improve access and change perceptions about computer science for students?

- First, we create computer science classes that succeed in minority-serving high schools. Jane Margolis at the the University of California Los Angeles has led the way in this strategy with the *Exploring CS* (<http://exploringcs.org>) that they have had adopted successfully in many California schools.
- Second, we change how we frame and describe computer science. Betsy DiSalvo at Georgia Tech has had success with her Glitch Game Testers project where she engages Black males with video game testing as a way of getting them to work with computing technology in a way that they find attractive. Her results are astounding, with the majority of her students moving on to study computing in higher-education.

Live coding can play a role in these strategies. Live coding gives an audience opportunity to see programming and to see a programmer, perhaps for the first time in their lives. They see programming in a live coding situation not as a solitary, asocial activity, but as a creative endeavor that creates a valuable product, music. Like Glitch, live coding can be engaging and draws students into exploring technology.

We need research to explore the value of live coding in changing perceptions about computer science, in engaging new students in exploring programming, and in the kinds of programming environments that are successful as both live coding environments and successful pedagogical tools. We need to explore how live coding can be adopted by a broad range of students. Teachers must be comfortable with the technology to invite it into their classrooms, so we need to explore how to make live coding inviting for teachers. Live coding can be a useful tool in improving access and changing perceptions about computing.

3.3 Live Coding, Software Engineering and Collaboration

Robert Biddle (Carleton University – Ottawa, CA)

License  Creative Commons BY 3.0 Unported license
© Robert Biddle

One of the topics of interest in our seminar was software engineering, and I was asked to give a brief overview to help our discussions. Our focus was collaboration, so I decided to focus on software engineering process, and especially on agile software development processes.

I began with the “waterfall” diagram from Royce’s classic paper [9] often used (unfairly [4]) as advocating a strict step-by-step process model. The steps in Royce’s diagram are requirements, analysis, design, coding, testing, and operations. I outlined the issues that arise in a strict flow, such as the need for early look-ahead to coding concerns, or the need to go back to design or analysis when requirements evolve or limitations are recognized. I also discussed the appeal of the waterfall, the vexing, but for some reason enduring, drift toward the strict flow between separated phases that can still be seen today. The role of Taylorism or “scientific management” has been recognized for some time [8] as affecting software process, and that may be the explanation.

This brought us to agile methods, and I introduced the basics of both Extreme Programming (XP) [2] and Scrum [10]. I emphasized the iterative nature of both processes, and explained the connection between iterative processes and collaboration, the way in which a dialog is fostered when work can be begun and then changed or refined subject to feedback and discussion. I emphasized the practices in XP and Scrum that facilitate this beneficial loop. In XP, for example, I discussed the “Planning Game”, having a “Customer On-Site”, the idea of “Test Driven Development”, how “Pair Programming” works, and the role of the

“Daily Standup”. For Scrum I discussed the roles, the “Product Owner”, the “Team”, the “Scrum Master”, and the flow from “Backlog” through “Sprint Planning” and “Sprint”.

I then moved on to discuss some topics where my research group has focused, typically through field studies of software development teams. The main topic I addressed was collaboration between programmers and interaction designers. Aside from embedded software, this collaboration is crucial to the success of software, but little has been said, either in academic work or in industry reports, about how this collaboration should work. I first discussed the work of Miller about interleaved processes [6], and the success it brought in their application domain. I then outlined our own findings. One common concept we identified was “alignment work”, where various practices, some formal, some less so, have the beneficial effect of cross-checking understanding between the groups involved in creating software [3]. Another important finding was the role of artefacts of various kinds in facilitating that alignment: stories, lists, sketches, and so on. These all serve to keep different viewpoints aligned, even as the software begins to take shape [2].

To conclude, I reviewed some of the challenges that arise in encouraging collaboration in software engineering processes. One I identified was that software development needs to connect to multiple other processes: business processes, organizational change, and so on. The nature of software is that it serves other purposes, and it can be difficult to link the software development process in to all the processes that are relevant. The other challenge that I addressed is that raised by other innovative ideas in software development. One, for example, is “software craftsmanship” [5], which emphasizes the technical skills and practices of programmers, and which I feel can de-emphasize the importance of collaboration with interaction designers, customers, and others involved in creating successful software. Another recently prominent idea is that of “lean” processes [7] that, inspired by success in manufacturing, emphasizes the elimination of waste. Like software craftsmanship, this is a good idea, but can be misunderstood: practices involving iteration and collaboration can too easily be seen as superfluous.

References

- 1 Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999.
- 2 Judith M. Brown, Gitte Lindgaard, and Robert Biddle. Collaborative events and shared artefacts: Agile interaction designers and developers working toward common aims. In *Proceedings of the 2011 Agile Conference, AGILE'11*, pages 87–96, Washington, DC, USA, 2011. IEEE Computer Society.
- 3 Judith M. Brown, Gitte Lindgaard, and Robert Biddle. Joint implicit alignment work of interaction designers and software developers. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design, NordiCHI'12*, pages 693–702, New York, NY, USA, 2012. ACM.
- 4 Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003.
- 5 Pete McBreen. *Software Craftsmanship: The New Imperative*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- 6 Lynn Miller. Case study of customer input for a successful product. In *Proceedings of the Agile Development Conference, ADC'05*, pages 225–234, Washington, DC, USA, 2005. IEEE Computer Society.
- 7 Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- 8 Hugh Robinson, Patrick A. V. Hall, Fiona Hovenden, and Janet Rachel. Postmodern software development. *Comput. J.*, 41(6):363–375, 1998.

- 9 W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE'87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
- 10 Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

4 Group Discussions

4.1 Critical Engineering and Software Tools

Roly Perera (*University of Edinburgh, GB*)

License  Creative Commons BY 3.0 Unported license
© Roly Perera

The title of this session was inspired by the Critical Engineering Manifesto, <http://criticalengineering.org>.

Codifying Practices in Tools

In human crafts, tools and practices co-evolve. Tools come to assimilate and embody the practices they support; software engineering is no exception. *Test-driven development* (TDD), for example, evolved as a methodology alongside the development of unit-testing tools which support TDD. When considering tooling for live coding, it may be useful to look at other programming domains where time plays a central role, such as animation. We may also look to traditional software engineering for ideas or jargon – such as a *spike*, a functionally “narrow but deep” prototype of a system. Moreover different performance scenarios (say teaching vs. public performance) may imply quite different tool requirements.

Since effective tools co-emerge with patterns of usage, programmers need to be able to “code without fear”, unhindered by worries about the consequences of exploratory actions. In current environments, many user actions are transient rather than persistent (i.e. leave no record of what was done), and this and other uncertainties may be a source of anxiety which undermines exploration.

Programming environments for live coding need to be open-ended and extensible, like Smalltalk’s *read-eval-print* loop (REPL). Open-endedness means the environment places as few constraints as possible on how it is used. Extensible means that code snippets and other reusable fragments of code – perhaps a library of synthesisers – can be made easily accessible from the programming environment. Such small, composable micro-features may also be a useful granularity for teaching and sharing.

Both performative live coding and other forms of live programming emphasise the importance of short feedback cycles. Anecdotally, any programmer knows that it is often hard to see how something works by just staring at code. But there is a tension here: if short feedback cycles alleviate cognitive load, then necessarily they also reduce the need for skills that manage that cognitive load. And we should not presume that these skills are unimportant to the creative process.

Researchers interested in tools and practices need small, focused research problems – *drosophilae* for live programming research. The form that these easily digestible problems might actually take remains unclear. A programme for live coding research may challenge

foundational assumptions about programming languages: whether languages are visual or textual, or even whether tools and languages can be cleanly separated.

New Models of Collaboration

New collaborative tools have the potential to bring live coding to a wider audience, develop communities of live coders, and make the whole endeavour more mainstream, through exposure.

Live coding systems can be less opaque and more communicative than traditional sequencing software like Cubase – more *authentic*, in the sense of conveying what is actually going on. This is due to the plurality of roles for source code in live coding. Primarily it is used by the performer to construct the performance, but importantly it is also often a key *part of the performance*. Observing how the code changes and its relationship to the music can elicit an aesthetic response just as can the auditory component of the music. Moreover it can help the audience appreciate simply that *it is possible* to create such a performance live, which they may not have previously considered. Finally, the code may also be a (partial) *record* of the performance, useful for teaching or other communicative activities after the performance. Thus as with normal programming, the comprehensibility of the code is important, although typically less so for the live audience, where comprehending everything which is going on may not be practical or even desirable.

The performative emphasis of live coding may be able to inform more traditional software engineering. For example music is often learned by observing virtuosity, as in a music masterclass, but this has only recently begun to happen with traditional programming. Online videos in which an expert programmer works through a problem are now common. Indeed performative videos are fast becoming a primary medium for disseminating both practical and theoretical knowledge: see for example the recreational maths videos of Vi Hart of the Khan Academy. Given this potential, recorded coding performances should perhaps be considered a first-class output – part of the “codebase”, even.

The internet is transforming not only knowledge dissemination, but collaborative activities in general. New collaborative systems tend to be flatter, decentralised (peer-to-peer) and distributed, rather than hierarchical, centralised and local. GitHub, Wikipedia, and StackExchange are just three paradigmatic examples of the “new school”. As new collaborative practices emerge, we should expect these to be supported and codified by new tools. The new structures of course also have shortcomings, just as more traditional collaborative regimes, such as orchestras, have their own unique strengths. Not only that, new structures that start out simple or democratic often become baroque or draconian, as some argue has happened to Wikipedia.

Although the internet democratises along some dimensions, it has the potential to alienate along others, such as gender and computational literacy. American scholar Henry Jenkins has written about the importance of new media literacy to life in a participatory culture. Because of the pace of change, age can be a powerful source of disenfranchisement. Conversely, young people often achieve a very high level of fluency with computer technology, as exemplified by the thirteen million Minecraft users worldwide.

Hero Culture in Programming

Even outside performing arts, communities have their celebrity “performers”. In software development, these are usually influential industry practitioners and commentators. Often their role is to *simplify a message*; such figures were instrumental in the adoption of object-

oriented programming and agile methods. In programming language research, and the academic community in general, gurus of this kind may be less important.

Another role for a luminary is to act as a “benevolent dictator” on a project, avoiding the pitfalls of design-by-committee. (Fred Brooks, in *The Mythical Man Month*, compared a development team to a surgical team, lead by a Chief Surgeon, although his analogy seems quaint by the standards of modern lightweight methods.)

Every field also needs its visionaries. Recently Bret Victor set out a compelling vision of programming, showcased not through working implementations but via a series of carefully crafted demos. In an influential essay “Learnable Programming”, he explicitly distanced himself from a simplistic construal of live coding as the live updating of the output of a program, emphasising instead the importance of being able to “see” all of the computation, and arguing that programming today is like painting with a blindfold on.

As with any performance, a live coding audience focuses their attention on the performer or ensemble. An algorave places the programmer–musician centre stage, as a traditional club night does a DJ. This focus on the virtuoso is potentially in tension with the collaborative and participatory potential of live coding. While some live coders feel that a performance should – in principle at any rate – be reproducible by a member of the audience, others are happy with a clearer distinction between virtuoso performer and audience member.

4.2 Algorithmic Complementarity, or the Impossibility of “Live” Coding

Julian Rohrerhuber (Robert Schumann Hochschule für Musik – Düsseldorf, DE)

License © Creative Commons BY 3.0 Unported license
© Julian Rohrerhuber

Origin and Motivation of the Concept

Live coding tries to closely superpose a contradictory pair of entities, both of which are implied in usual understanding of the term the term *program*: a structural description for something to happen in the future and the temporal unfolding of its consequences. By opening up the conventional black boxes in computer use, live programming apparently sets out with the promise to finally allow immediate and complete access or control. I have proposed the term *Algorithmic complementarity*¹ to explicitly name an easily overlooked but fundamental impossibility of making this superposition complete.

The concept of *complementarity* is derived in analogy to complementarity as it has been used in the philosophy of physics and psychology [1, 2, 3] where it refers to a number of phenomena (like subatomic particles or bistable figures) that have in common that their complete description requires two points of view which cannot be fully reconciled. I think the disadvantages of borrowing such a fairly loaded term from a different domain is outweighed by its potential to shine a different light on live programming. It also may point toward the theoretical relevance of this practise in a more general consideration of art and science, which is something that has always been an inspiration for me. [4, 5, 6]

For live coding, I propose that the fundamental impossibility of complete and transparent access to a programmed process is a positive constraint rather than a hindering obstacle.

¹ It was introduced last year at the International Symposium on History and Philosophy of Programming: Algorithmic complementarity. Some thoughts on experimental programming and the history of live coding. Cf. <http://www.computing-conference.ugent.be/file/12>.

Live coding demonstrates this impossibility in general, and finds local solutions to it, which results in the necessarily variety of existing systems.

Recounting the Dagstuhl Session on Algorithmic Complementarity

At the Dagstuhl seminar, we had one session on algorithmic complementarity, where we tried to better understand the core of the problem (and the conditions under which it ceases to be a problem). In the following, I'll try to recount these steps.

1. Let's assume a very simple algorithm that is formulated as a function of time, like $y = \sin(t)$, and that y has non-trivial consequences, like controlling a second, more interesting process.
2. At a certain point in time t_1 , the programmer realises that the outcome of this function should be different, e.g. that it should actually have double frequency.
3. She changes the code, e.g. to $y = \sin(2t)$.

A live coding system tries to correlate such changes in the description with changes in the running process, treating the code as its most accurate representation. So what should be the appropriate semantics of such a change? This depends on an interpretation of the formalism. There are a number of possibilities, which fall in two classes: state picture and causal picture.

- The *state picture* interprets the change as a change that happens to the state in which the running system is at that very moment (t_1). By consequence, it will either start the process $y = \sin(2t)$, setting $t_1 = 0$, or it may instead interpret the change as a parameter change of an assumed implicit parameter a , like in $y = \sin(at)$. Thus, the state picture takes the observable reasons into account which motivate a given change. The new code doesn't describe the system from its beginning, though: the causal relation between the process and the program text is fragmented.
- The *causal picture* is interested instead in maintaining the consistency of the description at all times. By consequence, it will retroactively interpret the whole past in terms of the change at t_1 . This means that one could restart the program from the beginning and at t_1 one would get the same result. The resulting state of the system, however, may not at all be related to the observation that led to the change in the first place – y may end up to be a very different value, and given that it is connected to some non trivial process, this may make the whole intervention meaningless. Maintaining the causal picture fragments the state picture.

In analogy with the corresponding phenomena in physics, it is argued that the two pictures cannot be reconciled in a single one.

4.2.1 Preliminary Conclusions

In such a way one can conclude (at least this is the hypothesis) that there is an incommensurability of two complementary interpretations of rewriting a part of a running program. As Roly Perera emphasized that when one unfolds all consequences of such a change and keeps all textual changes represented as changes,² a different set of problems appears: program and

² e.g. in a function like:

$$y = \begin{cases} \sin(t) & \text{if } t < t_1 \\ \sin(2t) & \text{otherwise} \end{cases}$$

result now are both fragmented. The resulting unfolding becomes an interesting navigational process, which in itself might not be representable on the same level.

Live coding is a non trivial activity anyhow, as in many cases (like sound synthesis), the relation between code and result is necessarily nonobvious – the resulting surprises or frictions with intuition or convention are a large part of its benefit (equally for learning, art, and science). So in a way, algorithmic complementarity is but one aspect of a much broader issue. It seems though as if algorithmic complementarity, if it really exists, is a fairly fundamental obstacle – an empty center perhaps, that prevents live coding from becoming a merely “technical” problem.

References

- 1 Atmanspacher, Harald, Thomas Filk, and Hartmann Römer. Complementarity in Bistable Perception. See [2], pp. 135–150.
- 2 Atmanspacher, Harald and Hans Primas (Eds.) *Recasting Reality. Wolfgang Pauli's Philosophical Ideas and Contemporary Science*. New York, 2009.
- 3 James, William (1890). *The Principles of Psychology. Volume One*. New York.
- 4 Rohrhuber, Julian, Alberto de Campo, and Renate Wieser. Algorithms today – Notes on Language Design for Just In Time Programming. In *Proceedings of International Computer Music Conference*, Barcelona, pp. 455–458. ICMC, 2005.
- 5 Rohrhuber, Julian. Implications of Unfolding. In Seifert, Uwe, Jin Hyun Kim, and Anthony Moore (Eds.), *Paradoxes of Interactivity*. Bielefeld, 2008.
- 6 Rohrhuber, Julian. Das Rechtzeitige. Doppelte Extension und formales Experiment. In Volmar, Axel (Ed.), *Zeitkritische Medienprozesse*. Berlin, 2009.

4.3 Tidal: Representing Time with Pure Functions

Alex McLean (University of Leeds, GB)

License  Creative Commons BY 3.0 Unported license
© Alex McLean

Despite being the most common noun in the English language, *time* can be quite difficult to talk about. Sometimes, the subject of time has been purposefully avoided in computer science, its discussion seen as a kind of anthropomorphism of pure mathematics, or simply a matter for optimisation. Over the past ten years, since the development of Live Programming as pedagogic and exploratory method [2], as live performance practice [1], and with the recent exploratory work of Bret Victor, the subject of time has been brought back on the agenda not only in computation but also in the act of programming itself.

A focus of one session was the music representation of Tidal, a Domain Specific Language (DSL) embedded in the Haskell language. Its representation is built upon the following principles;

Time is both cyclic and linear

In music we hear both cyclic repetitions and linear progressions, and so an expressive representation should allow us to work with both conceptions of time, at the same time. In other words, the representation should enable structure to be described at multiple timescales.

Pattern is both continuous and discrete

Our electronic computers, as fundamentally digital technology, naturally lend themselves to discrete representations. But we need not stop there; we experience the world, including music, through mental imagery as well as discrete events, and so our musical representations should allow for continuous shape as well as discrete sequences.

Live coding representations are not just designed for describing music, but also for thinking about, improvising with and composing music

It is not enough to be able to encode any piece of music theoretically, the language of composition needs to be adaptable to a particular style, and extremely quick to work with and explore.

Representations for composing music should be perceptually salient

That is, the encoding of pattern should be structured in a way that *potentially* be reverse-engineered in Human perception, just by listening to the output. That said, ambiguity is generally important in Music, being open to multiple possible interpretations by the creative listener.

In the following we focus on the first two principles, which are served through the following features of Tidal’s representation of time.

```
type Time = Rational
```

Tidal represents time simply as a monotonically increasing, rational number. This time value represents the current metric cycle³ number, so that a time value of “8/3” would be the point that is two thirds through the third cycle. This straightforward notion is enough to allow us to think about and manipulate time either in progressive linear, or repeating cyclic terms.

Tidal represents both continuous and discrete patterns with the same higher order type signature:

```
type Arc = (Time, Time)
type Event a = (Arc, a)
data Pattern a = Pattern {arc :: Arc -> [Event a]}
```

Simply put, a pattern is a function from time, to events. This allows pattern transformations to be expressed as combinators, which may operate on time separately from the contents of each event, or vice versa.

Because Time is a rational number with arbitrary precision, a Pattern cannot be efficiently represented and queried as a function from a single time value; we would have to sample a pattern through multiple queries to retrieve all the events, and depending on the rate of sampling that we choose, may miss some events. Instead, a Pattern is represented as a function from a time *range*, for which we use the term Arc, in sympathy with the notion of time as a cycle. In practice then, a Pattern is a function from a given Arc of time, to a set of events which occur during that arc. Furthermore, each Event is returned with its own Arc, representing when the Event begins and ends.

³ In musical terms, a *meter* is the underlying structure of a rhythm.

This representation works well for representing long, complex patterns. Firstly, it allows random access, so that we only need to calculate the particular section of a pattern we are currently interested in. Secondly, it allows patterns to be manipulated without calculating the events themselves, using a higher order combinator function.

Furthermore, by virtue of the use of Arc based queries, this representation works for notionally continuous as well as discrete patterns. For continuously varying patterns, we simply take the midpoint of the Arc of the query as the value to return.

In Tidal, the Pattern type is an instance of the Applicative functor and Monad classes, which allows them to be composed in a highly expressive manner, serving the latter two design principles listed earlier. Visual examples are provided in the paper “The Textural X” [3].

References

- 1 Nick Collins, Alex McLean, Julian Rohrer, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003.
- 2 Christopher M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.
- 3 Alex McLean. The textural X. In *Proceedings of xCoAx2013: Computation Communication Aesthetics and X*, pages 81–88, 2013.

4.4 Explicit vs. Implicit and External vs. Internal (Representations of) Time

Julian Rohrer (Robert Schumann Hochschule für Musik – Düsseldorf, DE)

License  Creative Commons BY 3.0 Unported license
© Julian Rohrer

Ignited by Alex McLean’s introduction of his live coding system Tidal, and by the excellent moderation of Thomas Green, there was a spontaneous session on the question of time: how is time represented in different systems and is there a fundamental difference in those representations? On an implementation level, timing can be a complicated issue, not least because it is equally relevant for layers of computation that concern the micro and the macro scale, which each may have very different constraints in terms of efficiency and expressivity.

Implementation itself wasn’t so much the topic of the discussion, however. What was more central was the way the variable *time* becomes part of the representation of the relation between a script and a process. It became apparent that:

1. some systems do not make explicit reference to a “state of time”, some make use of it excessively, even treating time as a first class object,
2. some systems locate the changing time state outside (usually a single value), some locate it inside (often multiplying the time references).

As the central domain of live coding are temporal phenomena and programs that unfold *over* time, this pair of differences, *explicit time* vs. *implicit time* and *external time* vs. *internal time* has a lot of ramifications in the trade offs between expressive power and expressive clarity. It is no wonder therefore, that the ontologies and computational aspects of time have been focus of numerous publications in the live coding community. Specific solutions were discussed at the seminar, such as cyclic time (which in a sense is an intermediary between changing time and static time), temporal recursion, knitting with time, and multiparadigmatic systems.

Also we exchanged knowledge about different ways of specifying sequences of events, in particular considering the difference between polymetric and polyrhythmic interpretations of a time step.

4.5 Stress and Cognitive Load

Alex McLean (University of Leeds, GB)

License  Creative Commons BY 3.0 Unported license
© Alex McLean

It goes almost without saying that live coding music in front of an audience can be stressful. Now that live coding is popular in conference presentations, as well as in educational settings, this issue of stress has wider significance. We formed a large breakout group to explore this issue from a number of perspectives.

Firstly, we noted that cognitive load is not necessarily to be avoided. When participating in music and other creative tasks, we enter a state of ‘flow’ [2]; deep engagement which requires a happy medium somewhere between frustratingly difficult on one side, and distractingly boring on the other. In other words, to support exploration and curiosity, we want *some* cognitive load, but not so much that things become too stressful to concentrate.

The Cognitive Dimensions of Notations framework [1] is well established in the Psychology of Programming field, and in our discussion we made good use of its terms in talking about the design trade-offs involved in cognitive load. The dimension of *hard mental operations* is one; live coding environments should be designed so that mental processing is targeted on the semantic rather than notational level. This relates to *closeness of mapping*; if the notation corresponds closely to the problem world, i.e. supports Domain Specific Language (DSL), then we are more able to engage the ‘right kind’ of cognitive load, and become more absorbed in programming as a live experience. Lack of *hidden dependencies*, low notational *viscosity* and of course facilities for *progressive evaluation* are other highly relevant design aims which can be articulated using the cognitive dimensions.

In conventional programming situations, the comprehensibility of a program is of great importance, but in live coding performances the requirements are different: we established that, in the opinion of most of the live coders present, the audience’s ability to comprehend the code as it was displayed was not paramount. Thus some of the aspects highlighted by the cognitive dimensions framework had a lesser role than in conventional programming. Nevertheless, the various languages used by live coders differ in many important ways, and the interfaces used to represent and interact with the code are similarly very varied. These large differences between probably have significant impact on how the code evolves in performance and on audience reception. For those interested in notational design *per se* detailed comparisons would be fascinating, but for most members of the live coding community, it seemed that there was more interest in what could be achieved with a particular tool than in abstract comparisons between tools.

We discussed behavioural differences of programmers when coding under the stress of tight time constraints. For example, whereas copy-and-paste is normally seen as bad programming practice, when a programmer only has a matter of seconds to effect a change, then it becomes a more attractive technique. This is especially true in live coding tasks such as improvised music; maintainability is hardly a factor if the code is discarded at the end of a performance.

There is some relationship between cognitive load, and ease of learning. The popularly

known “learning curve” of programming is related to the creative path of flow, and to be widely accessible, live coding environments need to support a good path for learners, including by supporting collaborative practices which provide social scaffolds for learning. Indeed, the possibility of learning during performance was raised; creating new functions during preparation, and exploring the possibilities of these functions in front of an audience.

A particularly interesting issue is that understanding your own code is not the same thing as being able to change it. In live coding of music, being able to quickly make a change, and then quickly adjust it in response to the musical result, requires knowing where you can make a change, and what kind of change you can make, but does not require fully understanding everything else about the code. One possible design response that arose is to make it easier to locate the section of code that is producing a particular aspect of the the sounds and patterns you can hear. This is to some extent being explored through visual environments such as SchemeBricks and Texture, but perhaps much more could be done to visualise data flow and functional transformations within the code environment.

References

- 1 Blackwell, Alan, and Thomas Green. Notational Systems – the Cognitive Dimensions of Notations framework. In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, ed. J. M. Carroll, 103–134. Morgan Kaufmann.
- 2 Nash, C. and Blackwell, A.F. Liveness and Flow in Notation Use. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME) 2012*, pp. 76–81.

5 Practical Implementation Session

5.1 Setting Up a Shared Temporal Frame

David Ogborn (McMaster University – Hamilton, CA)

License  Creative Commons BY 3.0 Unported license
© David Ogborn

In a session on Wednesday afternoon, the live coding performers in the group addressed the “infrastructural” issue of a common temporal frame shared via standard networking technologies. There are many systems (ranging from simple ad hoc constructions through to more elaborate, maintained and independent software projects) for synchronizing beats and other types of musical data. The infrastructural issue is not a lack of ways of sharing timing information, but rather an excess of such methods. Each individual, or group, or in some cases broader community, has one or more synchronization techniques. A lack of agreed upon protocols around these techniques hampers collaboration, especially when it is a case of co-performance involving multiple, distinct live coding software environments.

Much of the shared temporal frame session was devoted to exploring and hacking around the EspGrid software. EspGrid was developed by seminar participant David Ogborn in the specific context of a large, live coding laptop orchestra that uses multiple languages/-performance environments (the Cybernetic Orchestra at McMaster University in Hamilton, Canada). EspGrid responds to the multiplicity of performance environments by producing and maintaining synchronization elements independently of those performance environments (it is a separate application, developed in Objective-C and linked against the Cocoa or GNUstep libraries). This separation from the environments in which live coders perform allows

EspGrid to implement synchronization techniques of arbitrary complexity while shielding live coding performers from that complexity. It also allows different synchronization techniques to be tested and compared efficiently, as the specific synchronization technique can be changed independently of the code actions carried out by an individual performer.

While a common and relatively language-neutral software utility like EspGrid addresses some of the challenges of synchronization, it does not in and of itself abstract away the diversity of representations of time and metre. This was directly evident in the hacking that ensued during this session: the live coding performers, working very quickly, attempted to connect their habitual and preferred “internal” representations of time and metre to the public interface presented by EspGrid, with varying degrees of success. While the live coding performers performed this work of adaptation, the other seminar participants collected observations about the nature of the process unfolding before them.

Following the Dagstuhl seminar and informed by its results, a group of seminar participants have continued to work on the definition and dissemination of a common protocol for shared clocks and metres. Both the discussion at Dagstuhl and its later continuation have informed recent updates of the EspGrid software.

6 Live Coding: Some History and Implementations

6.1 Personal Accounts on the History of Live Coding

Renate Wieser (University of Paderborn, DE), Julian Rohrhuber (Robert Schumann Hochschule für Musik – Düsseldorf, DE)

License © Creative Commons BY 3.0 Unported license
© Renate Wieser and Julian Rohrhuber

In the following, we briefly sketch a few historical aspects of live coding, bearing in mind that such a description (besides being too short anyway) cannot be satisfactory – after all, as it is said, “the contemporary witness is the natural enemy of the historian”[2] How and when was live coding discovered? If this question that can be answered at all, it is from a personal perspective. Once recognised, however, live coding turns out to have never been entirely absent.

One marker for the recognition was, together with the first publication,[1] the first meeting *changing grammars*, organised at the University of Fine Arts of Hamburg (HfbK) in winter 2004. In a sense, this institution was a fairly natural environment for live coding: computers had already been an integral part of art education for a long time, with Kurd Alsleben as one of the protagonists of early computer art. Various cooperations with the computer science department of Hamburg University existed, as well as a number of computer labs with an emphasis on experimenting, network art and free software.

Certainly, this conference was a decisive moment for the form live coding has today. Firstly, it showed that the idea of rewriting programs at runtime wasn’t relevant only to a few individuals, but was a problem with many facets and faces and could lead to many different approaches. This variety has increased ever since and was an important topic at the Dagstuhl seminar. Secondly, it brought together two important understandings of the word “live”, which remain central today:

1. programming as *public* thought
2. reprogramming a program at *runtime*.

By delocating programming to unusual places like night clubs[3] and by deliberately entering the paradoxes of using programs to interact with programs, at that time, both aspects had been realised in a rough, radical, and experimental way. What “public” means could have been anything from a large concert to an informal conversation or even an abstract concept.

The further development of live coding in many ways went beyond any expectations at that time. Then already, however, we can recognise a tendency toward the unobvious, weak, hard to convey, which, together with exposing this practice to a partly uninformed public, includes a certain inclination to joke, failure and slapstick. Live coding could have been oriented more toward virtuoso culture, but it seems that the more conceptualist orientation has (gladly) remained till today. Also situating the live programming practice at disciplinary interstices, in particular between art, theory, science and education, was so common that it was hardly talked about – it wasn’t an explicitly transdisciplinary endeavour. The domain of sound, in its non-visual and irreducibly temporal character, was an important condition for this development. The founding of TOPLAP (one surprisingly non-temporary outcome of this meeting) led the way to a broader community and a consciously open definition of the practice and its terminology (including live coding, live programming, just in time programming, conversational programming, interactive programming).

Even if already recognised initially, in the subsequent years, at least two aspects of live coding came to the surface: firstly, the fact that many ideas had been already existing in computer science, but had been occluded by an over-emphasis of the computer as a tool simulation. After the peak of interactive programming in the late 1970s it was almost completely forgotten (while sometimes implicitly practiced) in mainstream computer science. When Adrian Kuhn at Dagstuhl reported from his recent live programming workshop that the participants were either quite young or relatively old, this may point toward the fact that the realisation of the relevance of live programming is indeed a retrospective one.

Secondly, it became rather obvious that the recognition and formulation of the central ideas – programming as public thought and programming as rewriting of running programs – would necessitate and give rise to a multiplicity of different approaches, languages and disciplinary contexts.

References

- 1 Nick Collins, Alex McLean, Julian Rohrerhuber, and Adrian Ward. Live Coding in Laptop Performance. *Organized Sound*, (8):321–330, 2003.
- 2 Wolfgang Kraushaar. Der Zeitzeuge als Feind des Historikers? Neuerscheinungen zur 68er-Bewegung. *Mittelweg 36*, (8):49–72, 1999.
- 3 Alex McLean. Hacking perl in nightclubs.

6.2 Extempore

Andrew Sorensen (Queensland University of Technology – Brisbane, AU)

License  Creative Commons BY 3.0 Unported license
© Andrew Sorensen

What makes Extempore reasonably unique in the current computer music landscape, is that Extempore is efficient enough, and general enough, to make its own audio-stack implementation available for on-the-fly code modification at runtime. The complete audio stack, right

down to the audio device driver callback, is written in Extempore and can be extended, modified, or completely swapped out, on-the-fly at runtime.

Extempore is a general purpose programming environment that hosts two programming languages, Scheme, and XTLang, and a standard library whose growing scope includes, 2D and 3D graphics, audio signal processing, mathematics, I/O and networking, concurrency, high performance computing, abstract data types, etc.

Extempore's Scheme interpreter is a derivation of the TinyScheme interpreter modified for real-time use. Extempore's Scheme interpreter was lifted, in whole, from the Impromptu programming environment, and in this sense is somewhat of a legacy component in the Extempore ecosystem. XTLang is a new, general purpose, high performance programming language designed for both the 'low-level' programming tasks often associated with 'systems' languages like C and C++, as well as the higher level programming often associated with 'scripting' languages. In many respects Extempore derives directly from C, supporting low level type expressivity, first class pointer semantics, manual memory management and a static type system. However, Extempore also borrows heavily from functional languages like ML, supporting first class closures, tail recursion, parametric polymorphism (aka reified generics) and type inferencing. Most importantly XTLang supports the same level of on-the-fly code compilation and runtime substitution commonly expected of so called 'scripting' languages. XTLang uses a homoiconic s-expression syntax similar to Scheme, and supports Common Lisp style macros.

Extempore's two languages are tightly integrated and share two of the environment's most important architectural features – a real-time scheduler for precise control over code start and execution times, as well as a flexible distributed concurrency semantics. These two qualities combine to provide the programmer with imperative control in both time and space. Providing imperative control over wall-clock-time and distribution-in-space makes Extempore well suited for tasks whose domain is the physical environment, with all of the causal consequences that this entails. Extempore's mantra is “real-time in real-time”.

As a live coding language, Extempore supports the high-level expressivity favored by many live coding environments, but also provides the low-level expressivity and high-performance required to code down-to-the-metal.

6.3 Fluxus

Dave Griffiths (FoAM – Kernow, GB)

License © Creative Commons BY 3.0 Unported license
© Dave Griffiths

Fluxus is a game engine designed for rapid prototyping and live coding of audio/visual 3D worlds. It's been used for live coding performance, immersive interactive games, architectural visualisations and teaching of music and programming concepts from children in the Brazilian rainforest to teenagers in Barrow in Furness in the UK.

Being built from the Racket language, a dialect of Scheme, Fluxus is an application of programming language research being carried out by the extensive academic community around that project.

Fluxus is also used as a foundation to support the research of new visual programming concepts for live coding performance. Betablocker is an interpreter for live coding assembler in graphical form with a game pad – prototyped in Fluxus and later developed as a Gameboy

DS game. Daisy Chain is a self modifying petri-net live coding language where the performer creates colourful dynamic graphs to sequence musical patterns. Scheme bricks is an interface for programming Scheme graphically, with close integration with audio for performer and audience feedback tightly connecting a bespoke code description and it's sonic results.

In recent years Fluxus has moved into browsers, mobile and games console technology, providing a diversification of live coding into new areas such as citizen science games and re-programmable tools for researchers working in remote locations.

6.4 ixi lang

Thor Magnusson (University of Sussex – Brighton, GB)

License  Creative Commons BY 3.0 Unported license
© Thor Magnusson

ixi lang is a live coding programming language whose interpreter is built in SuperCollider, thus giving access to the underlying power of that environment. However, unlike SuperCollider, the aim with the language is to create expressive constraints. Inspired by operator overloading in C++, the live coding systems of Alex McLean, and esoteric languages such as Whitespace and Brainfuck (see <http://www.esolangs.org>), ixi lang was designed as a high-level system affording certain types of musical patterns, but excluding others. As such, the system itself becomes a compositional form. Here, constraints inherent in the language are seen as providing freedom from complexity, yet defining a large enough search space for musicians other than the author to explore and express themselves. The language is very simple and intuitive for audience members. Code can be written that changes other code (and updates the code in the same document), which allows for complex structures and changes over time that are not directly called by the performers.

ixi lang affords a specific set of musical activities. It provides a scaffold for externalising musical thinking and, through its simplicity, attempts to ease the live coder's cognitive load. As a live-coding system, it goes further than most common live-coding environments in providing a simple, high-level platform for musical expression. As the system is written in SuperCollider, regular SuperCollider code can be written in the same document, allowing the user to tap into the extensive scope of SuperCollider itself. Learning the affordances of ixi lang as presented in its language constructs might take less than an hour, but getting an overview of the system's constraints can take many long sessions of practice.

6.5 SuperCollider and the Just In Time Programming Library

Julian Rohrhuber (Robert Schumann Hochschule für Musik – Düsseldorf, DE)

License  Creative Commons BY 3.0 Unported license
© Julian Rohrhuber

A significant technological threshold in sound programming was the moment when sound could be calculated faster than real time: while before the late 1990s, programs resembled factories that would produce sound files, suddenly, a programmer could construct a program that became something like a conversation partner in a musical situation. In such a way, both graphical and physical interfaces became central orientation points in programming.

This significance of this type of interactivity has also structured how the relation between the variable (parameters) and the static (the program) has been conceived of. Today, programming still defines itself mostly as a preparational task in which this relation between change and its boundary conditions is specified.

The programming language SuperCollider [1, 2] was an interesting intervention at this point – its granularity of description of acoustic processes was tuned in such a way that the separation between parameter and program became increasingly porous. Code became concise enough to resemble gestures, poems, or interfaces, so that it became thinkable that the separation between programming activity and music making might have merely been a conventional one that had been kept in place by the existing practice of setting up graphical user interfaces.

JITLib,[3] which has later become integral part of SuperCollider, set out to remove the remaining stumbling blocks which had so far hindered this idea to become an actual practice. While live coding already at that time was fairly easy to implement in a purely imperative world (where state simply is changed), in a mixed world between pure functions and objects, there were a number of problems to solve. The different parts of the library have the purpose to make it possible to experiment with the non-trivial relation between changes in the code to respective changes in sound.[4]

This experimentation has thus been made an integral part of various types of contexts of sound programming: learning, composition, performance, research, exploration. One of the outcomes of integrating the programming process into these activities is a very general insight: because programs are plans, or propositions for the future, changing such a plan in its unfolding cannot be immediate and transparent at the same time. From the beginning, Just-in-time-programming has therefore been a tongue in cheek cover term for the truth: a pro-gram is always too late.[5]

References

- 1 McCartney, James (1996). SuperCollider: a new real time synthesis language. In *ICMC Proceedings*.
- 2 McCartney, James (2002). Rethinking the computer music language: SuperCollider. *Computer Music Journal* (26), 61–68.
- 3 Rohrhuber, Julian and Alberto de Campo (2011). Just in time programming. In Collins, Nick, Scott Wilson, and David Cottle (Eds.), *The SuperCollider Book*. Cambridge, Massachusetts.
- 4 Rohrhuber, Julian and Alberto de Campo (2009). Improvising Formalisation: Conversational Programming and Live Coding. In Assayag, Gérard and Andrew Gerzso (Eds.), *New Computational Paradigms for Computer Music*.
- 5 Rohrhuber, Julian, Alberto de Campo, and Renate Wieser (2005). Algorithms today – Notes on Language Design for Just In Time Programming. In *Proceedings of International Computer Music Conference*, Barcelona, pp. 455–458. ICMC.

6.6 Overtone

Samuel Aaron (*University of Cambridge, GB*)

License  Creative Commons BY 3.0 Unported license
© Samuel Aaron

Overtone is an open source toolkit designed to facilitate the exploration of new musical ideas such as audio synthesis and sampling, instrument design, composition, live-coding and collaborative jamming [2]. It combines the SuperCollider audio synthesis engine, with Clojure, a JVM language with a specific emphasis on concurrency, immutability and functional programming.

Overtone supports multiple simultaneous users and provides first level language constructs to support the safe coordination of concurrent modifications in performance time. This allows new functionality to be hot-swapped over old. Overtone's support for lazy sequences enables infinite streams of events to succinctly model compositional structures such as chord progressions and recursive melodies. First class support for immutable data structures, as well as lock-free concurrency constructs, allow concurrent processing of incoming musical events, including storage and query, without having to block any running musical processes [1]. Finally, being hosted on the JVM provides Overtone with wrapper free access to a wealth of previously written Java libraries for interaction with external systems.

References

- 1 Samuel Aaron and Jenny Judge *New Possibilities For Social Digital Music-Making Arising From The Storage Of History* Proceedings Of The International Computer Music Conference, pages 228–235, 2012.
- 2 Samuel Aaron, A F Blackwell, Richard Hoadley and Tim Regan *A principled approach to developing new languages for live coding* International Conference on New Interfaces for Musical Expression, pages 381–386, 2011.

6.7 Republic: Collaborative Live Coding 2003–2013

Alberto de Campo (*Universität der Künste – Berlin, DE*)

License  Creative Commons BY 3.0 Unported license
© Alberto de Campo

Republic is an extension library for the SuperCollider language designed to allow highly collaborative forms of live coding. In a symmetrical network of participants, everyone can write and evaluate scripts which generate musical processes that can distribute sounds across all machines.

The library tries to minimize the difference between local and delocated events, i.e. between sound played on one's own computer and sound played on any other. Besides code and sound, a third layer of communication is chatting and shouting (i.e. hard-to-miss chat messages), which may serve to discuss the evolving performance, to make suggestions for next directions to take, send previously agreed-on cues, or sometimes try to distract each other.

Republic has been developed for the musical practice of the ensemble powerbooks_unplugged (since 2003), which evolved from a series of workshops called Warteraum (“waiting room”) which attempted to collide the concepts of granular synthesis and music in symmetrical networks [1]. It has since been used in academic teaching in several ways: the ensemble Republic111 at the Berlin University of the Arts has been performing based on the Republic

infrastructure since 2009, and the authors have given numerous workshops in many different contexts.

Over time, performance practice with Republic has tended towards more use of just-in-time programming as implemented in JITLib [2]: going from layers of independent, free-running shortish processes to fewer running processes and precise interventions into them allow more detailed shaping of the collaborative performance flow. Furthermore, the atomicity of JITLib-style programming has made reuse and adaptation of code lines by other participants easier, more likely to work immediately, and thus more rewarding and enjoyable. This can also lead to considerable uncertainties about causal relations of the concurrent audible processes (who of the delocalised players is responsible for which delocalised stream of sounds?), and about how to attribute authorship of the diverging versions of the scripts being written into and taken from the performance history [3].

It has turned out to be a useful introduction to programming for a wide range of artists: students of media arts, sound engineering, fine arts, music, and other fields often are encouraged to studying programming as an artistic practice they may want to integrate into their working repertoire, but very often show some reluctance, and at times even fear of coding. Reading very short programs that create a structure in time, hearing that structure unfold, and hearing how one's own changes to the code cause corresponding changes in the unfolding musical structure is a very immediate experience of the potential of programming.

By extension, such experiences allow deeper understanding of the role of programming in determining the behaviours of the countless devices and services that pervade today's technological societies. In fact, such experiences can illuminate the value of programming as a cultural technique that is both too important and too interesting to leave it to those only interested in the engineering side of the practice.

References

- 1 Julian Rohrhuber and Alberto de Campo. Waiting and Uncertainty in Computer Music Networks. In *Proceedings of the ICMC 2004, Miami*, 2004.
- 2 Julian Rohrhuber and Alberto de Campo. Just in time programming. In Nick Collins, Scott Wilson, and David Cottle, editors, *The SuperCollider Book*. MIT Press, Cambridge, Massachusetts, 2011.
- 3 Julian Rohrhuber, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl. Purloined letters and distributed persons. In *Music in the Global Village Conference*, Budapest, December 2007.

6.8 Sonic Pi

Samuel Aaron (University of Cambridge, GB)

License © Creative Commons BY 3.0 Unported license
© Samuel Aaron

Sonic Pi is a simple Raspberry Pi based environment designed to introduce basic computer science concepts through the creation of sounds. This approach aims to extend the creative opportunities for young Raspberry Pi users, engaging them through a direct emphasis on ownership and self agency within a programming context. Sonic Pi draws from many of the ideas within Overtone, presenting a text interface for manipulating the SuperCollider audio synthesis engine [1]. The text interface is implemented as a Domain Specific Language embedded within Ruby. Students interact with the language through a simplified text editor

that provides two basic controls: play and stop. Upon pressing the play button, the text is evaluated and sound is created. The stop button halts any currently executing process. This simple interaction cycle provides an extremely low barrier to entry for beginners, with their first program being as simple as typing `play 60`, then pressing the play button.

Sonic Pi has been developed through close collaboration with UK school teachers. It is accompanied by a scheme of work targeted towards introductory Computer Science within the newly created curriculum. Sonic Pi has also been used to explore the creative potential of technology within the arts sector. A project with artists, *Defining Pi*, explored the collaborative extension of the linguistic capabilities of Sonic Pi to directly support new art works. Current work focuses upon adding more liveness to the system, to enable the Raspberry Pi to be treated as a musical instrument, and in doing so bridge the computer science and music curricula.

References

- 1 Samuel Aaron and Alan F Blackwell *From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages*. FARM 2013, ACM

7 Personal Reflections

7.1 Clickety-Click: Live Coding and Software Engineering

Robert Biddle (Carleton University, CA)

License  Creative Commons BY 3.0 Unported license
© Robert Biddle

The seminar was an eye-opening experience for me, not in the sense of discovery, but rather of re-discovery, or perhaps more correctly of recognizing the importance of something so commonplace. At the workshop we considered various foci involving Live Coding: music performance, the humanities, computing education, and software engineering. I loved the music performances, and was inspired by the humanities perspectives, and the computing education expositions roused my missionary zeal. But I will not comment further on these. I am going to concentrate on the discussions on software engineering, as I saw them unfold at the workshop.

Agility and Collaboration

By using the term “software engineering”, we did not adopt any strong scoping: not any professional or political positioning. Instead we simply meant computer programming for some purpose. This activity is no longer new, and a few of us have seen it develop over half a century. And it’s hardly a niche activity, and indeed it is now the main work for millions of people, the foundation of giant organizations, and supports the infrastructure of much of the industrial world. But there is still uncertainty about how well we understand the nature of programming as an activity. When we must articulate that nature, we are still quickly drawn to exemplars: engineering, manufacturing, science, craft, and so on.

When the buzz about “agile” software engineering began, what was impressive was how it reflected actual experience. When Kent Beck first talked publicly about Extreme Programming in 1997, it felt refreshing to hear someone talk out loud about programmer experience, rather than invoking a model of what should work, based on some other activity. In truth, however, there has been such reflection throughout the history of programming,

and there is so much commentary on the activity of programming that it is hard to keep track. The material of programming allows many potential structures, and it seems the same is true for the activity. Moreover, in all the ecosystems involved, from technology to business, novelty itself appears to have value, even if only for the disruption it causes.

To begin our discussions on live coding and software engineering, we had a brief presentation on software processes, with a particular focus on collaboration and agile software engineering; the presentation is summarized elsewhere in this report (§3.3). The emphasis was on ideas in agile development that facilitate collaboration, such as pair programming, having the customer onsite, rapid cycles of development and acceptance, and willingness to change. Indeed, the subtitle of Beck's book [2] was "Embrace Change". In explaining how the collaboration between customers and programmers might work, one ideal often suggested was if the customer and programmer sat together, the customer could make suggestions and the programmer could implement them immediately for review and exploration. The customer might say "how about . . .", and the programmer would change the program: "clickety-click". This term, *clickety-click*, came to represent, in our discussions, the special character of software engineering exhibited in live coding.

The Blank Page

Hemingway called the blank page the "white bull". It stares at you, defiant, daring you to attempt writing. In live coding we reflected on how performers typically began with a blank screen. No program was running, and indeed no program existed. We made connections to doing this when teaching, or when demonstrating some programming to colleagues. The intent is to make the context clear and simple, to make it clear that there is no hidden process at work, that everything of interest will be introduced explicitly, and only that code explicitly introduced will be responsible for any behaviour observed. The aim therefore seems to be clarity, but also the establishment of common ground: my computer begins in the same state as yours. This emphasizes an important point for some performers: anyone can do this the same way. This supports the pedagogical aspect of live coding, and some performers articulated that perspective: they want their audience to understand that the process is visible and reproducible: anyone can make music just like this. As the magician says, there's nothing up my sleeve.

We reflected on elements of the magician's showmanship. It is manifestly impressive to audiences that music is created from a blank page. On the other hand, we also discussed the potential for skepticism. After all, a magician makes an effort to demonstrate a lack of hidden mechanisms. But then from nothing they produce scarves, or doves, or rabbits. It is not their intent to show that anyone can do this, but rather suggest to mystify, to demonstrate "magic". Might an audience feel live coding is making a similar claim? Or might they regard the feat as a different kind of demonstration, of mastery of the complicated skills and knowledge involved in programming? This might then resemble other kinds of music performance, where the audience does not learn how to perform similarly, yet does not ascribe the feats to magic, but nonetheless is appreciative of the live demonstration of skill and knowledge.

In considering the effects of the blank screen, we also acknowledged that no screen is ever really blank. There is much there: the computer itself, the programming environment, code libraries and frameworks, audio samples, network access, and all kinds of hardware, software, and content. So if the blank screen beginning demonstrates a beginning from zero, we have to ask what "zero" means. It can still, at least, mean common ground: an understanding that anyone else could start from the same place. If the audience is knowledgeable, then that

would work directly. But even if the audience is not so knowledgeable, then the claim might still have some impact: knowing that in principle they might be able to do the same. This kind of rhetorical device can also be misdirected, however: the Formula 1 car may be labelled “Renault”, but it is not the same Renault one can buy, and might have nothing at all in common, because naming rights can be bought and sold. So the live coding audience may still be skeptical, and this is worth further thought.

This discussion of skepticism raises some issues about how to present the nature of live coding. Live coding is about music and computing, art and science, whereas performers of “magic”, despite their appeal, are practicing deception. It seems important for live coding to be understood as eschewing deception. It is important for the audience to recognize this, and also for them to recognize the consequential and essential *vulnerability* of the live coding performer.

Small is Beautiful

In a live coding performance, the screen doesn’t stay blank for long. But while the performer may spend the entire performance typing, the size of the code never grows very large. Indeed, it often seems to stay about the size of the screen, meaning in the range of 20 to 50 lines of code. Sometimes it is longer, but still not very long. Of course, if the screen starts blank, the amount of code will be limited by the duration of the performance, and if a line of code takes a minute to imagine, type, and tweak, then an hour long performance only has at most 60 lines. Yet this still impresses, inspires, keeps an audience rapt. The brevity intensifies the effect, as does beginning from a blank screen, perhaps because the link between cause (the programming) and effect (the music) seems so clear, so immediate. I must admit, I’ve always liked small programs that do big things. The first programming language I learned, in 1968, was APL, known for enabling complex behaviour in a single line (albeit sometimes a long one). We take more care with the format and understandability of our code nowadays, but brevity still has appeal. Live coding reminds us of that appeal, and that power.

Small programs can do big things. Of course small programs can be made hard to understand – see the work on code obfuscation. But large programs are seldom really easy to understand. With a good supporting language and environment, small programs seem to have advantages in ease of creation, comprehension, correctness, adaptability, and manageability. This doesn’t always match up well with ease at the user interaction level: we discussed those early Apple Macintosh programs that were so appealingly easy to use, but concealed complex code in Macintosh Pascal. Will we ever resolve this conflict: can software be simple at both the user and the developer level?

But in live coding there seems to be an emphasis on keeping the operational elements more visible. There seems to be an explicit intent to show the workings while demonstrating their results. We discussed how this was essential when the program or the command line was the only way to interact with computers. While having a higher barrier to understanding, this leads to a deeper understanding. We reflected on how it later became commonplace for computer literacy to become a more superficial kind of literacy. One appeal of live coding, we thought, was the return to that earlier age. Some people even referred to those who grew up in the GUI era as “lost generation”, who learned how to use computers without learning how they worked. We also discussed how, on the other hand, graphical interfaces can also be used to expose inner workings: Alan Blackwell talked about his Palimpsest language [5], and Dave Griffiths demonstrated the visual language he uses in his live coding performances (Scheme Bricks) [8].

Reduce, Reuse, Recycle

In a live coding performance, the program is not only created live, but it is then changed over and over again. The program may be small, but it is not static. It seems there are many kinds of changes that are common. One is accretion, where code is added to create more complex behaviour. Indeed, even the initial creation from a blank screen might be seen this way, where the blank screen indicates a running program that simply does nothing; when more code is added, so the behaviour begins to be visible, or rather audible. Sometimes code is taken away, simplified, or deleted, and the music becomes less complex. And this is how some performances end, returning to an empty program and silence. Other performances ended when a change broke the code – those are also important because they show the performance risks are real, and familiar!

Sometimes code is created by being copied from one place and pasted in another. In live coding, the code typically comes from elsewhere in the current program. In real life, code comes from other local files and sometimes from the wild Internet. (This approach has been called “Scrapheap Programming”, where search engines help find code to be repurposed, adapted, and fitted into place [9], but this is not typical a live coding performance.) Other functionality typically involves simpler textual operations: we type keystrokes and enter text, can delete and insert, search and replace, and so on. (Of course, there are more refined approaches to reuse, and indeed the history of programming and programming languages can be seen as efforts to advance ways of supporting reuse [3].) We make macros or functions with parameters and call them; we create classes and instantiate them; or objects and clone them; we create frameworks and populate them. And all these also work with accretion: we add code, we rename, we split, we merge, we refactor. And we did see some of this in live coding, albeit on a small scale.

An important thing to notice here is the interplay between the programming language and the environment. They are not really separate, and how well they fit together is a significant usability issue. This important relationship between notation and environment is explored in the Cognitive Dimensions of Notation framework by Green, Petre, and Blackwell, [6, 7, 4]. In live coding this is all at work: the language, the environment used together. The small size of the programs raised some interesting questions. With such small size, do we need such powerful tools? Or are the programs only small because of the power of the tools? I think I saw evidence for both arguments. At the workshop, Thomas Green also emphasized the need to support secondary notation, “visual cues which are not part of formal notation”: interestingly we saw little use of that in live coding.

Reflecting on how the code develops and changes in live coding, we discussed the potential role of version control systems such as Subversion or Git. As well as helping manage code in different versions and from different programmers, these tools also make it possible to try out code ideas with the easy ability to change one’s mind. In this way they provide a kind of safety net, like an “undo” facility, which in turn encourages exploration. Some of the performers explained that they did use such systems, and one suggested it feels like it allows “programming without fear”.

Time and Tide

All programs can change what they do. They do one thing, then another, they select which of several things to do, and they do things over and over. They change what they do depending on input, whether static data or live input. With techniques like reflection, they can change their own code. All of this can happen rapidly.

In live coding, we see a different kind of change. The programmer changes the code while the program is running, based on their human perception of time. The changes happen continually, and always on the human scale of time. It would have been possible for the programmer to create the program up-front to cause the same behaviour, and indeed that would more closely resemble conventional programming. In live coding this is not done. Instead of complex coding done entirely before execution, live coding specifically means coding during execution. But it means more: the coding is part of the execution. The programmer is literally part of the machine.

In one of our sessions at Dagstuhl it was suggested that what performers did was tweaking. It often did seem like that: making small changes to get something just right. Except that it never stayed just right: or “just right” changed as the performance progressed. It was exploratory programming, and the direction of the exploration changed as the performance progressed. The tweaking *was* the performance. The tweaking, the act of seeking “just right” and then moving on: that was the experience being intentionally shared between the performers and the audience.

Live Free or ...

A live coding performance is compelling, and we discussed the experience as being reawakened to an almost forgotten sense of what programming can be. Live coding speaks to the programming soul. For more conventional software engineering, this raises questions about why this happens and what it suggests for improving the status quo. Our speculative answers to these questions come from the points above. To begin with the last point, much comes from the sharing of the experience of exploration.

As with any live performance, there is an anticipation, a tension, a continual awareness of uncertainty. Some live performances are like tightrope-walking, and the uncertainty is whether the performance will be successful. More relevant to live-coding is the uncertainty about what direction the performance will take. And most relevant is where there is a sense that the performers are communing with each other and with the audience. Having a role in creation, or co-creation, engenders a heightened awareness of what happens. Espen Aarseth suggests this is why games form a kind of “ergodic” medium, because one must work for the results [1].

If the main effect in live coding is the shared experience, it is also important to recognize that the other characteristics all support this main effect. To react is to change, and to react fast is to change fast. To support that, therefore, it is important to be able to change software fast. That means we need the tools, the language and the environment, to make that happen. The mechanisms involved, copy-paste, calling functions with parameters, refactoring, need to be at our fingertips. To making the scoping clear, it helps if our code is small, and to make sure that scoping is evident, there is even an advantage in starting from a blank page.

It is this experience that is possibly the key lesson of live coding for software engineering. In particular, if agile software engineering is about collaborating and about willingness to change, then the kind of shared experience shown in live coding is just what is needed. All kinds of techniques lead in the same direction. In interaction design, low-fidelity prototypes allow us to try things out and change them fast, in software engineering, test-driven development and automated testing, also lead to rapid feedback. It’s what made the first spreadsheet software so compelling, because the automatic and rapid recalculation encourages explorations that asked “what if?” It reminds me of Shneiderman’s articulation [10] about direct manipulation: “rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.” Tanimoto wrote reflectively on this effect and its implications, calling it “liveness”

[11]. It has taken decades, and it seems both necessary and worthwhile to keep looking at this.

To work together we need to be able to make changes fast in response to ideas, we need to be able to experience their effects. How fast? *Clickety-click*. That fast.

References

- 1 Espen J Aarseth. *Cybertext: perspectives on ergodic literature*. JHU Press, 1997.
- 2 Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999.
- 3 Robert Biddle, Angela Martin, and James Noble. No name: just notes on software reuse. *ACM Sigplan Notices*, 38(12):76–96, 2003.
- 4 Alan F Blackwell. Ten years of cognitive dimensions in visual languages and computing: Guest editor’s introduction to special issue. *Journal of Visual Languages & Computing*, 17(4):285–287, 2006.
- 5 Alan F. Blackwell and Ignatios Charalampidis. Practice-led design and evaluation of a live visual constraint language. Technical Report UCAM-CL-TR-833, University of Cambridge, Computer Laboratory, May 2013.
- 6 Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
- 7 T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.
- 8 Alex McLean, Dave Griffiths, Nick Collins, and Geraint Wiggins. Visualisation of Live Code. In *Proceedings of Electronic Visualisation and the Arts London 2010*, pages 26–30, 2010.
- 9 James Noble and Robert Biddle. Notes on postmodern programming. In *Proceedings of the Onward Track at OOPSLA*, volume 2, pages 49–71. Citeseer, 2002.
- 10 Ben Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983. cited By (since 1996)359.
- 11 Steven L Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

7.2 Cultural-Epistemological Aspects of Live Coding

Geoff Cox (Aarhus University, DK)

License © Creative Commons BY 3.0 Unported license
© Geoff Cox

Many of the introductions on the first day of the workshop suggested similar ideas that I addressed in my talk: in celebration of messiness in programming (Biddle); in esoteric and ad hoc language development (Blackwell, Brown, Aaron); and examples like SmallTalk or Scratch (Green, Griffiths, Kuhn); histories of interactive/conversational programming and the collaborative dimension (Rohrhuber, Romero); in emphasising real-time or just-in-time programming (de Campo, Rohrhuber, Sorensen); and the importance of the time dimension of programming and execution (Magnusson, McLean, Perera); as well as more conceptual concerns with issues of control and computational thinking, references to defamiliarisation and new (non-human) aesthetics, and even craft traditions – techne or poeisis (Rohrhuber, Wieser). Indeed live coding embraces the unexpected, the accident and an “aesthetics of

failure” (perhaps even what some commentators these days refer to as the “post-digital” condition).

Live programming is the more accepted term in the communities of software engineering and programming language development. Robert Biddle developed the idea of software development and language design through a short history of extreme programming and Scrum methods, and with examples of agile programming, to stress the significance of environments where the machine and programmers are writeable (e.g. executable scratchpad or xiki, a wiki with executable code) and the shift back to text-based languages (command line rather than GUI). It is interesting to note historical comparisons here and possibilities to conceive of programming as breaking with correctness and allowing for the programmer to reflect on how they think and feel while programming. The body is more in the frame, and together with the running program produce an embodied conception of software as a whole.

How do we understand live coding in terms of its genealogy (e.g. TOPLAP), the development of the field, the community that has emerged? This is not simply a development method but something more. Does it constitute a field or discourse? Where does it emerge from in terms of existing disciplines (computer science or music)? What do the analogies to interactive or conversational programming reveal?

Temporality is also clearly a key concern that live coding is able to highlight: a central issue in both computing and music-making but perhaps a neglected one at least in computational cultures. With live coding the source/plan and its execution run at the same time. A focused discussion ran on the issue of time with reference to the paradox that time is both linear and cyclical (cf. §4.3). Does procedural logic allow for a different understanding of time, and perhaps a different understanding of the creative process, where the programmer and program are running in the very dynamic processes of history (cf. Benjamin’s “Jetztzeit”/nowness)?

This brings me back to my interest to further emphasise the political and epistemological aspects (someone mentioned Turkle’s “epistemological pluralism”): that live coding produces new knowledge through artistic and technical experimentation; and open up analogies to other notation practices more broadly (where scores and execution have resonances in terms of action and agency). Social context also is important, inasmuch as new formations of publicness can be perceived – here making reference to the writings of Hannah Arendt and her understanding of the political. In *Two Bits*, Chris Kelty argues that the free software movement is an example of what he calls a “recursive public”, extending Arendt’s definition of a public through speech and action, to incorporate technical and legal infrastructures. Publicness is constituted not simply by speaking, writing, arguing, and protesting but also through modification of the domain or platform through which these practices are enacted. Live coding seems to fit the description and recombinations of human and non-human (or more-than human) bodies, and new materialisms.

7.3 Dagstuhl: Live Coding and Memorable Discussions

Dave Griffiths (FoAM – Kernow, GB)

License  Creative Commons BY 3.0 Unported license
© Dave Griffiths

Our seminar included people from the fields of Software Engineering, Computer Science Education as well as plenty of practising live coders and multidisciplinary researchers.

Discussion was wide ranging and intense at times, and the first job was to sufficiently

explain what live coding actually was – which turned out to require performances in different settings:

1. Explanatory demo style live coding: talking through it as you do it.
2. Meeting room coffee break gigs: with a closely attentive audience.
3. The music room: relaxed evening events with beer and wine.

So Dagstuhl’s music room was immediately useful in providing a more “normal” live coding situation. It was of course more stressful than usual, knowing that you were being critically appraised in this way by world experts in related fields! However it paid off hugely as we had some wonderful interpretations from these different viewpoints.

One of the most important for me was the framing of live coding in terms of the roots of software engineering. Robert Biddle, Professor of Human–Computer Interaction at Carleton University put it into context for us. In 1968 NATO held a “Software Components Conference” in order to tackle a perceived gap in programming expertise with the Soviet Union.

This conference (attended by many of the “big names” of programming in later years) led to many patterns of thought that pervade the design of computers and software – a tendency for deeply hierarchical command structures in order to keep control of the arising complexity, and a distrust of more ad hoc solutions or any hint of making things up as we go along. In more recent times we can see a fight against this in the rise of agile programming methodologies, and it was interesting to look at live coding as a part of this story too. For example it provides a way to accept and demonstrate the “power to think and feel” that programming gives us as humans. The big question is accessibility, in a ubiquitously computational world – how can this reach wider groups of people?

Ellen Harlizius-Klück works with three different domains simultaneously – investigating the history of mathematics via weaving in ancient Greece. Her work includes live coding, using weaving as a performance tool – demonstrating the algorithmic potential of looms and combinations of patterns. Her work exposes the hidden shared history of textiles and computation, and this made a lot of sense to me as at the lowest level the operations of computers are not singular 0s and 1s as is often talked about, but actually in terms of transformations of whole patterns of bits.

Mark Guzdial was examining live coding through the lens of education, specifically teaching computer science. The fact that so many of us involved in the field are also teaching in schools – and already looking at ways of bringing live coding into this area, is noteworthy, as is the educational potential of doing live coding in nightclub type environments. Although here it works more on the level of showing people that humans make code, it is not a matter of pure mathematical black boxes – that can be the ground breaking realisation for a lot of people.

Something that was interesting to me was to concentrate on live coding as a specifically musical practice (rather than also a visual one) as there are many things about perceiving the process with a different sense from your description of it that are important. Julian Rohrhuber pointed out that “you can use sound in order to hear what you are doing” – the sound is the temporal execution of the code and can be a close representation of what the computer is actually doing. This time-based approach is also part of livecoding working against the notion that producing an “end result” is important, Juan A. Romero said that “if you’re livecoding, you’re not just coding the final note” – i.e. the process of coding is the art form.

In terms of a school teaching situation sound is also powerful, as described by Sam Aaron, live coder and creator of Sonic Pi. A child getting a music program to work for the first time

in a classroom is immediately obvious to everyone else – as it is broadcast as sound, inspiring a bit of competition and ending up with a naturally collaborative learning experience.

It is impossible to cover all the discussions that we had. It was a great opportunity to examine what live coding is about now in relation to other practices, where it came from, and where it might go in the future.

7.4 Live Coding, Computer Science, and Education

Mark Guzdial (Georgia Institute of Technology, US)

License © Creative Commons BY 3.0 Unported license
© Mark Guzdial

Most of the attendees were live coders, but there were a number of us others who helped explore the boundary disciplines for live coding. The seminar explored the ramifications and research potential of this activity.

Robert Biddle was there to lead discussions about the software engineering implications of live coding. On the one hand, live coding feels like the antithesis of software engineering, or as one attendee put it, “an ironic response to software engineering.” There is no test-driven development, no commenting, no development of abstractions (at least, not while live coding), no exploration of user needs. On the other hand, live coding (not necessarily with music) can be an important part of exploring a space. One could imagine using live coding practices as part of a conversation with a user about needs and how the programmer understands those needs.

Geoff Cox led a conversation about the humanities research directions in live coding. Geoff has a particular interest in labor, and he pointed out how live coding surfaces hidden aspects of the labor in modern society. While computing technology has become ubiquitous in the developed world, few people in our society have ever seen source code or a programmer. What does it mean for an audience to see an embodiment of a programmer, to see the labor of generating code? What’s more, the audience is seeing code doing something that is not normally associated with people’s notions of what code is for – making music. How does this change the audience’s relation to the computing technology? The notion of programming-as-performance is an interesting and novel way of thinking about computing practice, and in sharp contrast to stereotypical perspectives of programming.

Thomas Green, Alan Blackwell, and others from the PPIG (Psychology of Programming Interest Group) community pointed to the notations that the live coders used. I’ve drawn on Thomas’s work on the cognitive dimensions of programming and the implications of our programming representations since I was a graduate student. The language constraints for live coding are amazing. Live coders tend not to use traditional development languages like C++ or Java, but instead work in Scheme, Haskell, and a variety of domain-specific languages (like SuperCollider) – often building their own implementations. Live coders need languages that are expressive, provide for the ability to use techniques like temporal recursion, are concise, and (as one live coder put it) “let me live code at 2 am at a dance club after a couple of beers.”

I was there to connect live coding to computing education, and learned about many connections from the seminar. I am fascinated by the humanities questions about introducing source code and programmers to a technologically-sophisticated audience that probably never saw the development of code. Also, I am interested in the value of rapid feedback (through

music) in the performance. Does that help students understand the relationship between the code and the execution?

A Playful Live Coding Practice to Explore Syntax and Semantics

Three of the nights of the Dagstuhl Seminar on Live Coding included performances. Several of these combined live coders with analogue instruments (guitar, piano, cello, and even kazoo), which was terrific to watch.

I found one of their practices fascinating, with real potential for the computer science classroom. Alex McLean introduced it as “Mexican Roulette”, because they first did it at a live coding event in Mexico City. Live coders take turns (the roulette part) at a shared computer connected to speakers at the front of the room.

- The first live coder types in some line of code generating music, and gets it running. From now on, there is music playing.
- The next live coder changes the code any way she or he wants. The music keeps playing, and changes when the second coder then evaluates the code, thus changing the process. Now the third coder comes up, and so on.
- If a live coder is unsure, just a few parameters might be changed.
- If a live coder makes a syntax error, the music continues (because the evaluation that would change the process fails), and the next coder can fix it.
- If a live coder makes a mistake (at one point, someone created quite a squeal), the next live coder can fix it. Or embellish it.

What I found most promising about this practice is that (to use Briana Morrison’s phrase for this) nothing is ever *wrong* here. The game is to keep the music going and change it in interesting ways. Responsibility for the music is shared. Mistakes are part of the process, and are really up for definition. Is that a mistake, or an exploration of a new direction? This activity encourages playing with syntax and semantics, in a collaborative setting. It relies on the separation of program and process – the music is going, while the next live coder is figuring out the change. This could be used for learning any language that can be used for live coding.

Live Coders Challenge Computer Science to Think About Expression Again

Live coders think about and talk about expression, as evidenced from the conversations at Dagstuhl. They build their own languages and their own systems. They talk about the abstractions that they are using (both musical and computational, like temporal recursion), how their languages support various sound generation techniques (e.g., unit generators, synthesized instruments, sampled sounds) and musical styles. If you look at the live coders on the Dagstuhl Seminar participant list, most of them are in music programs, not computer science. Why are the musicians more willing to explore expressive notations than the computer scientists?

Lisp is alive and well in live coding. I now have a half-dozen of these systems running on my laptop. Overtone is a wonderful system based in Clojure (<http://overtone.github.io/>) Andrew Sorensen’s Impromptu was in Scheme, as is his new environment Extempore. Andrew said that he could build 90% of this in Impromptu, but the low-level bits would have to be coded in C. He was not happy with changing his expressive tools, so he created Extempore whose lowest level parts would be compiled (via LLVM) directly to machine code. Andrew went to this effort because he cares a lot about the expressiveness of his tools.

Not everything is s-expressions. Thor Magnusson's *ixi lang* is remarkable. I love how he explores the use of text programming as both a notation and a feedback mechanism. When he manipulates sequences of notes or percussion patterns, whatever line he defined the sequence on changes as well.

Tidal from Alex McLean is a domain-specific language built on top of Haskell, and his new Texture system creates more of a diagramming notation. Dave Griffiths has built his live coding environment, Fluxus, in Racket which is used in Program by Design and Bootstrap CS education projects. Dave did all his live coding at Dagstuhl using his Scheme Bricks, which is a Scratch-like block language that represents Scheme forms.

Education Research Questions Around Live Coding: Vygotskian and Non-Constructionist

I saw four sets of computing education research questions in live coding. These are unusual research questions for me because they are Vygotskian and non-Constructionist.

Live coding is about performance. It is not an easy task. The live coder has to know their programming language (syntax and semantics) and music improvisation (e.g., including listening to your collaborator and composing to match), and use all that knowledge in real-time. It is not going to be a task that we start students with, but it may be a task that *watching inspires* students. Some of my research questions are about what it means to watch the performance of someone else, as opposed to being about students constructing. I've written before about the value of lectures (<http://computinged.wordpress.com/2010/07/27/in-defense-of-lecture/>), and I really do believe that students can learn from lectures. But not all students learn from lectures, and lectures work only if well-structured. Watching a live coding performance is different – it's about changing the audience's *affect* and *framing* with respect to coding. Can we change attitudes via a performance?

Vygotsky argued that all personal learning is first experienced at a social level. Whatever we learn must first be experienced as an interaction with others. In computing education, we think a lot about students' first experience programming, but we don't think much about how a student first sees code and first sees programming. *How can you even consider studying a domain whose main activity you have never even seen?* What is the role of that coding generating *music*, with cultural and creative overtones? The social experience introducing computing is important, and that may be something that live code can offer.

Here are four sets of research questions that I see:

1. *Making visible*. In a world with lots of technology, code and programmers are mostly invisible. What does it mean for an audience to see code to generate music and programming as a live coder? It is interesting to think about this impact for students (does it help students to think seriously about computing as something to explore in school?) and for a more general audience (how does it change adults' experience with technology?)
2. *Separating program and process*. Live coding makes clear the difference between the program and the executing process. On the first day, we saw performances from Alex McLean and Thor Magnusson, and an amazing duet between Andrew Sorensen at Dagstuhl and Ben Swift at the VL/HCC conference in San Jose using their Extempore system. These performances highlighted the difference between *program* and *process*. The live coders start an execution, and music starts playing. Meanwhile, they change the program, then re-evaluate the function, which changes the process and the music produced. There is a gap between the executing process and the text of the program, which is not something that students often see.

3. *Code for music.* How does seeing code for making music change student's perception of what code is for? We mostly introduce programming as engineering practice in Computer Science class, but live coding is pretty much the opposite of software engineering. Our biggest challenges in Computer Science Education are about getting students and teachers to even *consider* computer science. Could live coding get teachers to see computing as something beyond dry and engineering-ish? Who is attracted by live coding? Could it attract a different audience than we do now? Could we design the activity of live coding to be *more attractive* and accessible?
4. *Collaboration.* Live coding is a collaborative practice, but very different from pair programming. Everybody codes, and everybody pays attention to what the others are doing. How does the collaboration in live coding (e.g., writing music based on other live coders' music) change the perception of the asocial nature of programming?

I will end with an image that Sam Aaron showed in his introduction, a note that he got from a student in his Sonic Pi class: "Thank you for making dull lifeless computers interesting and almost reality." That captures well the potential of live coding in computing education research – that activity is interesting and the music is real.

7.5 Algorithms, Live Patterning and the Digital Craft of Weaving

Ellen Harlizius-Klück (University of Copenhagen, DK)

License © Creative Commons BY 3.0 Unported license
© Ellen Harlizius-Klück

From the perspective of the humanities, in my case: the history of textile technology, live coding offers interesting perspectives on questions I generated in my research on patterning textiles in antiquity. In both cases, craftsmanship is a category that is to define within the research. The sociologist and philosopher Richard Sennett introduced the idea of software as craftsmanship in a first chapter of his book on the craftsman. As we heard from Robert Biddle, this idea was already strong in software engineering, long before, but not comparable with what live coders understand, when they use this term. Interestingly, Sennett subtitled one of his chapters "Ancient weavers and Linux Programmers". However, he rarely says anything about what they have in common.

Margaret Boden's book on Artificial Intelligence is more instructive in this respect, describing algorithms as executing knitting notations. But described like this, even a cooking recipe is an algorithm and the knitting example does not really convince of a similarity of knitting and computers executing programmes on a structural level.

My interest lies just in this: demonstrating that weaving was a digital and even dualistic craft from the very beginning and that the patterning principles can tell us a lot about the prediction or impossibility of prediction of patterns even if we fully control the production process. I am therefore looking for tools to code textile patterns, and live coding with such a tool would enable me to show the production of patterns and their changes on the fly when demonstrating my topics to an audience. It was interesting to see some coders already working with the ideas of threads, textures (Alex McLean) and patterned weaves (Dave Griffiths). Live coding of textile patterns at the moment can only be done with very complicated programmes that are designed for the use of modern looms and for designers who, on the level of sketching, work with vector graphics instead of the dualistic and discrete patterns that are the basis of weaves. Coding patterns on a principal level can help me in

showing that such patterns are not flat but an interference effect of a three dimensional structure and a special colour order of its elements. Live coding of textile patterns also makes the algorithmic nature and the involvement of number ratios visible.

From the seminar I learned that live coding could be a way of thinking publicly, as artwork today mainly does. And it can change our perception of people working with computers as it was demonstrated by the quote that Renate Wieser presented: “Art removes objects from the automatism of perception in several ways.”[1] Today programmes project the impossibility of liveliness into the future. We need a counterbalance against this attitude, a counterbalance that does not reject programming as a whole but develops possibilities of liveliness in a surrounding of codes and programmes. Also here, doing live coding as an art form, can contribute. As Renate Wieser said: Artists are stopping habitual processes and not starting new habits (which science does). Live coding therefore can be a possibility of exploring the relationship between liveliness and automation. Live coding also implies a critical perspective on engineering as the most influential language of our times. And with live coding of pattern weaving I try to make room for women in the history of computer and software development.

References

- 1 Victor Shklovsky. *Art as Technique*. 1917, 10 f.

7.6 A Few Notes from the Seminar

Julian Rohrhuber (Robert Schumann Hochschule für Musik – Düsseldorf, DE)

License  Creative Commons BY 3.0 Unported license
© Julian Rohrhuber

The forgotten dimension of programming which turned out to be only half forgotten

Some historical notes: In the first years, live coding developed within a specific mixed context between experimental arts, club music, philosophy, and interdisciplinary scientific research. It grew where people worked together as musicians, as artists, but also as researchers in interdisciplinary science projects. It was immediately part of a teaching curriculum, or rather its self-recognition as something special came from learning exchanges, workshops, collaboration, conversation art, lecture performance.

In such a way, computer science education has been its driving motor from the beginning – the fact that the forms and topics of teaching didn’t resemble those of informatics courses shouldn’t obscure the insight that this was collaborative learning efforts that centred on topics like higher level functions, polymorphism and networking protocols. From the programming languages involved, this practice inherited knowledge (both tacit and explicit) from a long history of programming languages.

A note regarding the visibility of programming: the necessity to expose the programming activity, as well as to see the programming activity as a part of either a performance or the computational process was far more obvious in this transdisciplinary context than it was for those who had contemporary training in computer sciences. As it was said a couple of times during the discussion in the seminar, the abstract and elusive character of the domain (namely sound in the art and sound as a means to explore scientific data) made it both more fruitful to do live coding (or just in time programming, as we usually call it) and more essential to achieve anything interesting in the first place. As far as education concerns: this

context made it very obvious that learning itself requires the experience of programming, of thinking a domain (like sound) through the medium of a program.

Another point implied in the discussions in the seminar: programming always has a domain, a domain that is addressed (to some degree) in computer science and computer science education. In the case of pure theoretical computer science, this domain is mathematical, or formal structures. In many other cases, the domain is a mixed field of knowledge, that may involve fields as diverse as music, astronomy or accounting. Invariably, however, programming means speaking doubly, to a machine and to other humans. This is I think what Robert Biddle showed to be a constant challenge in the relation between programmer and domain expert, and also between programmers. In how far is it this involvement in a domain through the activity of programming itself, that live coding makes evident?

Maybe it could be summarised as follows: Live coding was discovered (or, if to a degree rediscovered) by computer science and by computer science education – but not in their respective disciplinary institutions. Reintroducing it now to those institutions, the question is: should this happen as a transfer of a technical tool-idea or should this also transfer at least parts of the philosophical, methodological, and institutional backdrop that was necessary for this idea to proliferate? Both seem possible futures.

Chess players

Taking up Geoff Cox' thought: live coding brings to the surface the social and labour dimension of programming: programs, like texts, are written, often in a tedious and vacillating process, involving decisions both rational and aesthetic, sometimes practical and ad hoc. Showing writing (in the broad sense) reveals the “agency behind” a program, the agency that allows to make plans rather than just to act. In such a way, live coding shows abstraction in the form of delegation (this may be just a sort of steering, if it is a more immediate translation, this may also be a pro-programming proper in the sense of involving in changing planned setups). On the other hand, what becomes visible also is that this labour is not reducible to any core of direct action or intention, it is rather already in itself a mediated process, involving human actions, but actions which only make sense in the context of a system, a programming language, its abstract but sensual semantics, its syntactic resistance and fluency, its ability to speak about itself, to be understood. In such a way, in our context, van Kempelen's fake chess robot is half the truth: inside the hollow case, the dwarf is sitting, certainly, but what makes the dwarf a chess player? Only the context of the chess system, and the fact that the audience understands this context makes the automaton so fascinating.

In a discussion with Thomas Green, which couldn't be continued, two understandings of the term “plan” came up in the context of a chess game: Thomas referred to a plan as something that a human skilled player has in mind. I pointed out that in a program, contrary to the chess board, plans are laid out and become readable as plans. I should add now that in a chess constellation, a skilled player can certainly decipher possible plans; but it would arguably defeat the purpose of the game if all this were explicit. Thomas disagreed: what we see in a program is not a plan. The sociological and psychological semantic dimension of a plan may indeed be something entirely different than the formalised aspect of a plan which we can read in a program. Nevertheless, if the agency of planning should really play a role in live coding (and I think it does), it is a particular indistinguishability between those aspects that define live coding as something different than playing an instrument or a chess game. The fact that this agency of planning is partly obscure, potentially to the audience but also to the programmer – for instance when a program becomes readable only retroactively – is one of the reasons why it is so interesting.

Participants

- Sam Aaron
University of Cambridge, GB
- Robert Biddle
Carleton Univ. – Ottawa, CA
- Alan Blackwell
University of Cambridge, GB
- Andrew R. Brown
Griffith Univ. – Brisbane, AU
- Luke Church
University of Cambridge, GB
- Geoff Cox
Aarhus University, DK
- Alberto de Campo
Universität der Künste –
Berlin, DE
- Thomas Green
University of York, GB
- Dave Griffiths
FoAM – Kernow, GB
- Mark Guzdial
Georgia Inst. of Technology, US
- Ellen Harlizius-Klück
University of Copenhagen, DK
- Shelly Knotts
Birmingham, GB
- Adrian Kuhn
University of British Columbia –
Vancouver, CA
- Thor Magnusson
University of Sussex - Brighton,
GB
- Alex McLean
University of Leeds, GB
- David Ogborn
McMaster Univ. – Hamilton, CA
- Jochen Arne Otto
ZKM | Center for Art and Media
Karlsruhe, DE
- Roly Perera
University of Edinburgh, GB
- Julian Rohrhuber
Robert Schumann Hochschule für
Musik, DE
- Juan Gabriel Alzate Romero
Hochschule für Musik –
Karlsruhe, DE
- Uwe Seifert
Universität Köln, DE
- Andrew Sorensen
Queensland University of
Technology – Brisbane, AU
- Jan Kees van Kampen
Robert Schumann Hochschule für
Musik, DE
- Renate Wieser
Universität Paderborn, DE

