# Technical Dimensions of Live Feedback in Programming Systems

JOSHUA HOROWITZ, University of Washington, USA

While live feedback plays an important role in many interactive programming systems, its design space remains largely unmapped, making it difficult to discuss and build on the wide range of designs explored by past systems. As a first step towards establishing this map, we present six dimensions that can be used to characterize and evaluate live feedback in programming systems: granularity, reactivity, velocity, moldability, bidirectionality, and materiality.

## 1 Introduction

Programming systems are called "live" when they provide feedback on the dynamic behavior of programs as they are edited [1, 18, 19, 22, 24, 30]. A broad range of interactive programming systems can be classified as live, from augmented code editors like Projection Boxes [18] and programming-by-demonstration systems like Wrangler [15] to dynamic media like Boxer [5]. Developers of live programming systems have high hopes for the value feedback can provide, including increasing accessibility for less experienced programmers, easing comprehension while reading code, and opening up new exploratory workflows [22].

Live feedback is not a single straightforward mechanism, or even a one-dimensional spectrum from "less" to "more live". Rather, it is complex and multifaceted, varying across many dimensions between systems. Unfortunately, the community of live-system researchers and designers lack a common framework for discussing and analyzing these variations – an instance of the more general lack of theory for programming *systems* (as opposed to programming *languages*) [14].

From our experiences working on live programming systems and our review of prior systems, we have identified six dimensions that characterize live feedback: ***granularity***, ***reactivity***, ***velocity***, ***moldability***, ***bidirectionality***, and ***materiality***. (See Figure 1 for a graphical summary of these dimensions.) This paper presents these dimensions as a first step towards understanding this design space. We expect there are many more dimensions left to articulate, and perhaps more helpful reconfigurations of these six dimensions. We are excited to evolve these dimensions alongside the live-programming community, building a shared map of the territory we are exploring together.

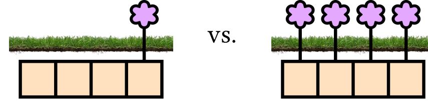
## 2 Related Work

We are aware of few attempts to "map out" the space of live-programming systems, as we do here. The most commonly-cited map is Tanimoto's "levels of liveness", a four-level hierarchy of liveness in visual programming systems [24] later extended with two additional levels [25]. Quoting from the later paper, these six levels are: (1) Informative, (2) Informative and significant, (3) Informative, significant and responsive, (4) Informative, significant, responsive and live, (5) Tactically predictive, and (6) Strategically predictive. While a variety of projects have made use of these levels for guidance and evaluation [1, 3], we have found it difficult to work Tanimoto's hierarchy into our map of feedback in live-programming systems. Levels 1 & 2 do not require any live feedback at all, and instead focus on the representation of code itself as a visual structure, which we consider to be orthogonal to liveness. While the move to level 3 does invoke the kind of live feedback we are concerned with, the distinction between levels 3 and 4 is about reactivity to external event streams, which has not emerged as central in our work. Levels 5 & 6 once again diverge from our interest in feedback, as they refer to the programming system anticipating the programmer's future actions and synthesizing new code automatically. We suspect some of the

Author's Contact Information: Joshua Horowitz, University of Washington, Seattle, WA, USA.

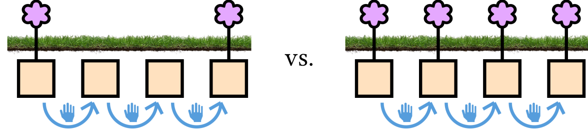
**Granularity**

How deeply into the structure of a program is feedback provided?



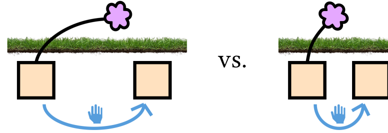

**Reactivity**

How often are changes to a program reacted to with feedback?



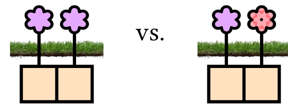

**Velocity**

How quickly is feedback available?



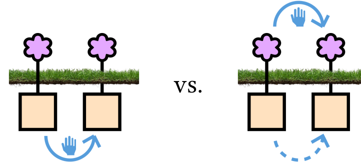

**Moldability**

How can feedback be shaped to reflect domain-specific meaning?




**Bidirectionality**

Can programs be edited by acting on feedback?




**Materiality**

Is feedback a side effect, or part of the critical path of computation?

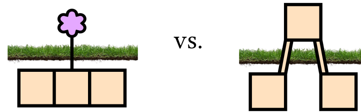
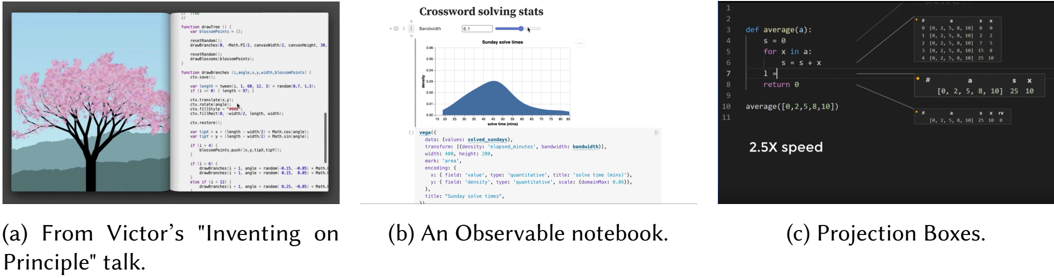



Fig. 1. An overview of the dimensions discussed in this paper and their iconic depictions, showing flowers of feedback sprouting from otherwise-invisible programs underground.

trouble we have had incorporating the "levels of liveness" into our work comes from its structure as a single linear hierarchy. We have found more fruitful to see the design space of liveness as multidimensional, naming specific dimensions and mapping out their interactions, as we do here.

We are inspired in our approach by Technical Dimensions of Programming Systems (TDoPS), a framework of dimensions for programming systems in general: "integrated and complete set[s] of tools sufficient for creating, modifying, and executing programs". [14] TDoPS refers to live programming in a number of places, especially the dimension of Feedback Loops (§4.1.1), where Jakubovic et al. observe how live feedback helps minimize the gulf of evaluation [13]. Much of our work here can be seen as a closer examination of the Feedback Loops dimension, teasing out sub-dimensions within it.




(a) From Victor's "Inventing on Principle" talk.

(b) An Observable notebook.

(c) Projection Boxes.

Fig. 2. From low to high *granularity*.

Several other papers look at liveness from a birds-eye view. Rein et al. [22] perform a literature survey to examine how researchers employ the term "live programming", and how this relates to adjacent terms like "live coding" and "exploratory programming". Horowitz & Heer [12] carefully delineate "liveness", which they characterize as about dynamic feedback, from the overlapping but distinct category of "richness", which is about how programs themselves can be presented in visual, domain-specific forms. Along the way, they taxonomize a number of "approaches to liveness" which are helpful in our following discussion.

## 3 Dimensions

In this section, we introduce and briefly explore the six dimensions summarized in Figure 1.

### 3.1 Granularity: vs

How detailed is the feedback in a live system? Specifically, how deeply into the internals of the program does the system provide visibility? This is the dimension of *granularity* of feedback.

We can illustrate this dimension with three systems that vary along this axis, shown in Figure 2. First is a demo from Bret Victor's "Inventing on Principle" talk [27]. The user edits code on the right-hand side of the screen. As they do so, the output on the left instantly updates – a clear example of live feedback. However, this is the coarsest possible granularity: while the overall behavior of a program is made visible and responsive, the internals of the program (intermediate values, control flow, etc.) remain a black box. This level of granularity is typical of "liveness outside of code" [12].

Next is an Observable notebook[1]. Like other computational notebooks (and spreadsheets), Observable lets a programmer break up their code into "cells". Division into cells provides convenient sites for visibility: live values flowing between cells can be visualized alongside the cells. This gives Observable a higher granularity, typical of "liveness between textual code cells" [12].

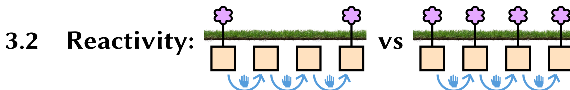
However, as Horowitz & Heer argue in their work on Engraft, the "flat", static structure of cell-based systems prevent visibility from extending into "nested structures like functions, loops, or conditionals" [11]. This motivates efforts to push all the way to the most fine-grained end of the granularity spectrum, "liveness within textual code / structure editors" where feedback is threaded through constructs of the programming language itself. We illustrate this category with Projection Boxes [18], though it includes many other systems, including the eight code editors surveyed in Rauch et al.'s "Babylonian-style Programming" [21].

[1] https://observablehq.com/platform/notebooks

With the granularity spectrum established, we can better explain the visual language we developed for dimensions of feedback, shown in Figure 1 and in section headings. Our guiding metaphor is that, without special efforts towards liveness, the steps of a program are hidden from view "underground". Liveness makes "flowers" of feedback sprout from these underground steps, revealing them to the programmer. Varying the different dimensions changes the structure and behavior of this blossoming in different ways. For instance, granularity is about how much live visibility extends into the parts of programs – lines of code, functions, notebook cells, etc. – not just the whole. So on the left (low granularity), we only have one flower coming up, because we only see final output. But on the right (high granularity), there are more flowers sprouting up from more parts of the program, creating a more continuous trail from input to output.

A finer granularity would seem to be advantageous, by providing more feedback to the programmer and leaving fewer places where they must tediously and delicately imagine in their mind what the computer is doing. However, excessive feedback can be distracting and reduce information density, diluting code with irrelevant details. Careful visual and interaction design is needed to ensure feedback doesn't get in the way. For this reason, effective liveness may in fact be more a challenge of information design than one of programming-language design.

### 3.2 Reactivity: vs



How often are changes to a program responded to with feedback in a live system? This is the dimension of *reactivity* of feedback.[2]

As an example of moving along this spectrum, consider Observable. Observable is a reactive notebook, which re-runs a cell automatically whenever its code or an upstream dependency value changes. In this way, it provides more reactive feedback than Jupyter [17], which requires cells to be re-run manually, even if their dependencies change. However, a code change in Observable only takes effect when it is saved manually or its text editor loses focus, not on every keystroke. In this way, its feedback is less reactive than Natto[3], which can re-run cells on every keystroke.

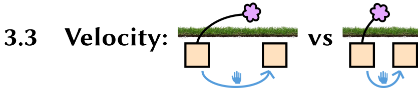
To depict reactivity of feedback in our visual language, we introduce a depiction of *edit time*, with blue arrows marked with hands representing edit actions by the user. These make the x-axis into a timeline, with the four blocks representing different states of a program, rather than different parts of a program as they did before. On the left, several edits can occur without new output feedback becoming visible, but on the right, every edit provides instant feedback.

Higher reactivity would seem to advantageous, by providing feedback to a programmer more quickly. With reactive feedback, programming begins to feel more like aiming a water hose than aiming a bow-and-arrow, to use a striking metaphor from Hancock [6]. However, over-reactive feedback can be dangerous. Code usually moves through invalid states on the way from one valid state to another. If these invalid programs are indiscriminately executed, this can make the programmer's experience sluggish, produce meaningless visual output, and possibly even trigger damaging side effects. Even if only feedback from valid states is shown, there is a risk (as with granularity) that excessive feedback may distract the user.

Several systems have taken ingenious approaches to obtaining the advantages of high change granularity without its pitfalls. JavaScript REPLs in some web browsers evaluate expressions on every keystroke, using special engine features to abort "eager evaluation" when side effects are

[2]Remarkably, Henderson & Weiser [8] identified this dimension in their paper on the live (*avant la lettre*) system VisiProg„ using the term "change granularity" for "the size of the change that VisiProg must detect before re-executing".
[3]https://natto.dev/

detected.[4] Hazel [19] uses a structural editor so that every editor state is meaningful and can produce helpful feedback. More prosaically, some cell-based systems (including Natto and Hex[5]) allow the user to select how often each cell should be re-evaluated – on each keystroke? only when run by hand? – thereby allowing the user to customize change granularity on a cell-by-cell basis.

### 3.3 Velocity: 

How quickly, after an edit is made, is feedback available to a programmer? This is the dimension of *velocity* of feedback.

Velocity stands out from the other dimensions in that more velocity is seemingly always better. A system designer would always prefer a faster system over a slower one. The question is instead one of engineering: how fast can feedback get?

The simplest way to update feedback after a program is edited is to run the edited program from scratch. As programs become larger, this can become prohibitively expensive. *Incremental computation* [20] provides techniques for avoiding this problem by only running the necessary parts of programs after an edit. Live-programming systems often turn to incremental computation to make fast & frequent feedback possible.[6]

The most well-known technique for incremental computation is data-flow computation (a.k.a. “reactive programming”), in which a program is understood as a network of small computations with data flowing from outputs to inputs. When one of these computations changes, only computations “downstream” of it must be invalidated and recomputed, while the rest of the network can be preserved. Data-flow computation is most easily implemented when a program is already structured as a flat network of cells, as is the case in spreadsheets (the most famous data-flow system) and reactive notebook systems like Observable and Marimo[7].

Work on incremental computation emerging specifically from the context of live programming includes Ziegler et al. [31]’s “patch-recon” system, which updates program output to reflect changes to code without full re-runs, and Horowitz & Heer [11]’s Refunc, which adapts React’s memoization primitives to incrementally evaluate nested live programs.

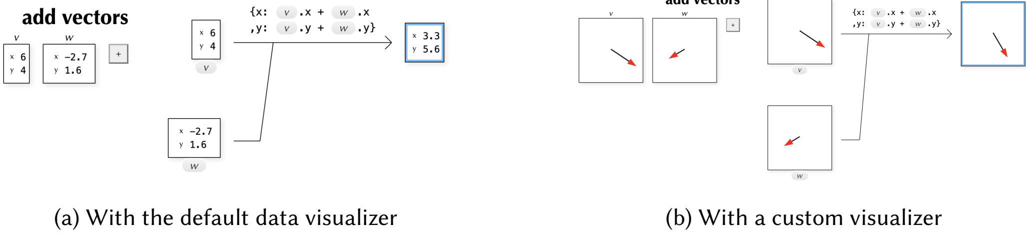
In some cases, even with incremental computation, it may not be possible to maintain a high velocity. Some computational tasks, such as analyzing large datasets or processing large media files, are inherently expensive and may take seconds, or days, to run. In these cases, a live-system designer confronts difficult design challenges: How can a system integrate feedback into a program editor when the feedback may be out of date, referring to an older version of the program? How can a user comfortably control the execution of their program as they make edits, when automatic execution might slow their computer to a halt or spin up supercomputing-cluster jobs?

### 3.4 Moldability:

How can feedback be shaped to reflect domain-specific meaning? This is the dimension of *moldability* of feedback.

Live-programming systems are committed to the maxim *“show the data”* [28]. But for data to be useful to a programmer, it has to be presented in a way that is meaningful in the programmer’s

[4] https://developer.chrome.com/blog/new-in-devtools-68#eagerevaluation
[5] https://hex.tech/
[6] Henderson & Weiser [8] (cited earlier for their coinage of “change granularity”) defined “recomputation granularity” as “how much re-execution is required in response to a change” – in other words, how incremental the system is.
[7] https://marimo.io/




(a) With the default data visualizer

(b) With a custom visualizer

Fig. 3. Live values in PANE can be shown with domain-specific visualizers, a form of *moldability*.

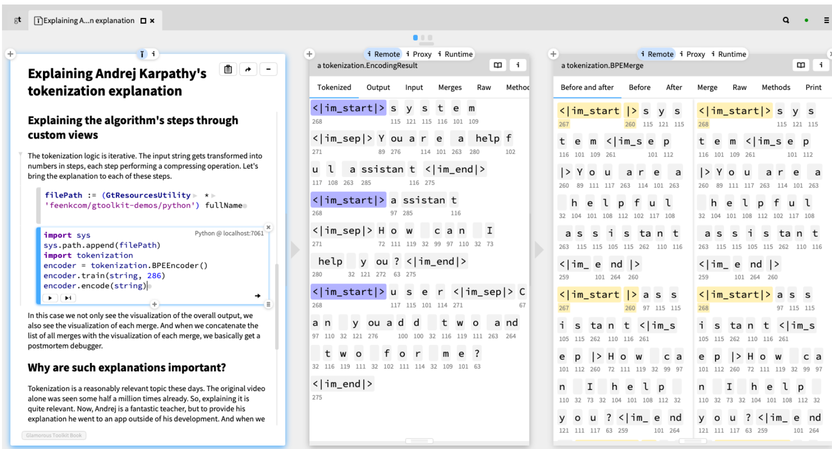
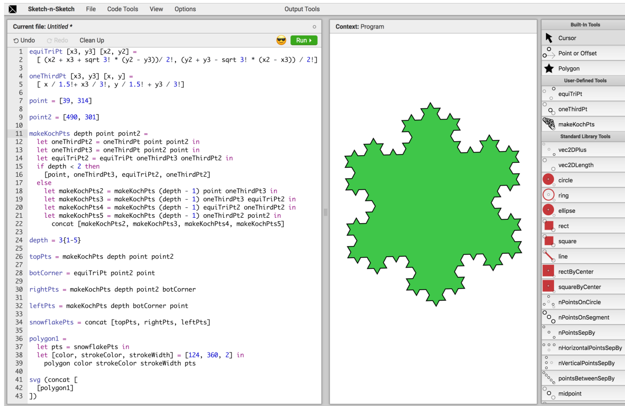



Fig. 4. The Glamorous Toolkit in use editing a tokenization algorithm. The output of different stages of the algorithm are displayed with custom visualizers, exhibiting *moldable* feedback.

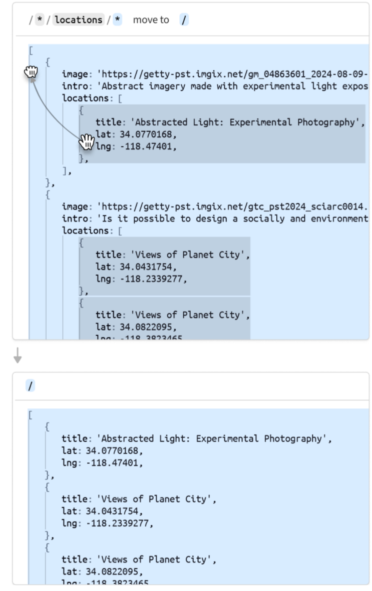
specific context. This caveat is especially relevant for systems that strive towards general-purpose applications. For instance, PANE is a visual programming system which shows intermediate values in small panels [9]. PANE does its best to show these values legibly, as seen in Figure 3a. However, as PANE works on arbitrary JavaScript data, this display is necessarily generic. Although vectors have a familiar and powerful geometric representation, all PANE sees is a generic object like `{x: 6, y: 4}`, so that is the only structure it can show. Fortunately, PANE's feedback is *moldable*: PANE lets programmers plug in special visualizers that define how to display values matching certain patterns. Figure 3b shows the PANE interface after a visualizer is added to show vectors as arrows.

Feedback is *moldable* when a programmer can change its appearance to reflect the needs of their particular use case. We draw this term from Chiş et al. [2] and ensuing work on the Glamorous Toolkit[8]. Figure 4 shows how the Glamorous Toolkit allows inspectors panels to be customized to show data-types specific to a project. The Clerk notebook system also supports creating custom viewers and in fact advertises itself as offering "Moldable Live Programming for Clojure"[9].

[8]https://gtoolkit.com/
[9]https://clerk.vision/

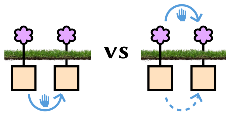


(a) Sketch-n-Sketch




(b) Sculpin

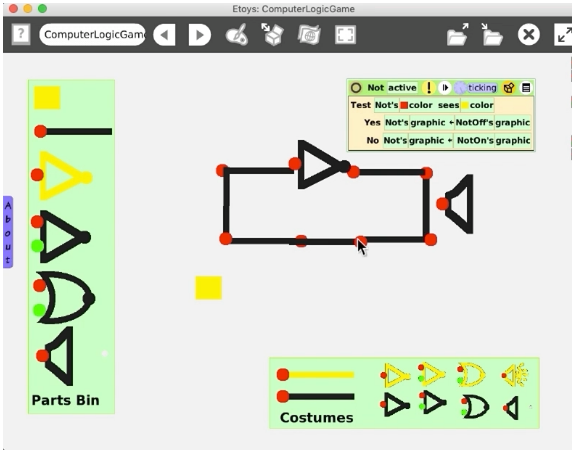
Fig. 5. Two systems exhibiting *bidirectionality*.

### 3.5 Bidirectionality: 

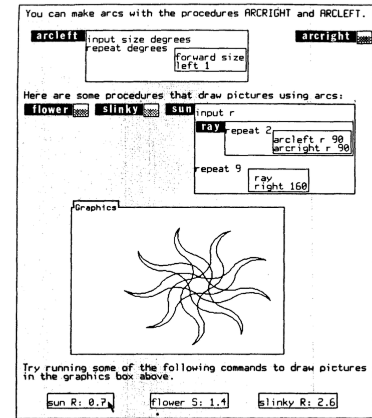

Can programs be edited by acting directly on feedback? This is the dimension of *bidirectionality* of feedback.

While most live systems present feedback for read-only inspection, referring the programmer to a source program to make edits if needed, some make feedback itself into a handle by which further edits can be made. At first glance Sketch-n-Sketch [7] (Figure 5a) looks like a typical code / live preview interface, where edits to the program in the left pane result in updates to the output graphic in the right pane. However, in Sketch-n-Sketch the live output on the right is also editable by direct manipulation. Edits to this output are transformed into edits on the textual program, which is ultimately the primary representation of the program. This is the paradigmatic arrangement for bidirectional live feedback.

As another example, Sculpin [10] is a programming-by-demonstration [4] system for transforming JSON. Figure 5b shows how a drag-and-drop operation on a source selection can create a flattened structure. Direct-manipulation operations like these are added to Sculpin's timeline as they are performed, so chains of operations can be used as reusable programs. In this way, the live feedback Sculpin provides acts as a handle for defining future steps in the program, just like in Sketch-n-Sketch. The term "bidirectionality" applies more loosely to Sculpin than to Sketch-n-Sketch, since Sculpin represents its underlying program in a DSL that is not amenable to direct edits, rather than in an ergonomic general-purpose programming language like Sketch-n-Sketch does. All the same, they share what we take to be the defining characteristic of bidirectionality: transforming edits on outputs into edits on programs.

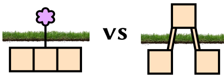



(a) A circuit simulator in eToys




(b) Boxer

Fig. 6. Two systems exhibiting a high degree of *materiality*.

### 3.6 Materiality: 


Is feedback a side effect, or part of the critical path of computation? This is the dimension of *materiality* of feedback.

The typical image of a live-programming system is one where live feedback *augments* a pre-existing program, as in Projection Boxes (Figure 2c). This means the feedback is in some ways "epiphenomenal" – if it were removed, the program would still make sense and keep on running. But certain programming systems entangle program and feedback more deeply. In these systems, visibility is not tacked on as a secondary consideration. Rather, the system is designed from the ground up to be made of entities that are visualized.

In thinking about this dimension, we have been particularly inspired by the example of a circuit simulator Alex Warth constructed in eToys [16, 29], shown in Figure 6a. In this simulator, each circuit component looks at its input dots to see what colors is underneath them – yellow or black – and then it uses that color to determine whether it should itself turn yellow or black. That means the thing humans are seeing, that makes the program's behavior visible, is actually the thing driving the program. The view is not constructed as an epiphenomenal visualization of underlying abstract data. Instead, the view is along the "critical path" of dynamics. Previously, our icons showed flowers popping up to reveal computation happening underground. But in a system with truly material feedback, the computation itself extends above the surface, and no flowers are needed.

This dimension has strong resonances in the history of computational media. Boxer (Figure 6b) is based on a principle its creators call "naive realism", which states that "users should be able to pretend that what they see on the screen is their computational world in its entirety" – no data should be hidden away from the visible world [5]. The developers of Self followed a similar principle when building the UI for Self [23]. In a later reflection, they summarized this principle as "the thing on the screen is supposed to be the actual thing" [26].

Achieving liveness through materiality is perhaps the most radical option available to a system designer. Traditional computer science tends to hold that data should be represented in mathematically elegant forms like algebraic data types, and that "views" should be strictly separated from

"models". Following these precepts will lead to a low level of materiality. But by departing from these disciplinary norms, materiality opens up intriguing possibilities with a profoundly different feel than conventional programming systems. In a system with material feedback, a program sees and works with the same things the programmer sees, and the programmer is no longer a second-class observer.

## Acknowledgement

We thank LIVE 2024 Workshop participants for helpful feedback on an earlier version of this work.

## References


[1] M.M. Burnett, J.W. Atwood, and Z.T. Welch. 1998. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254)*. IEEE, 126–133.

[2] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. 2015. Towards moldable development tools. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 25–26.

[3] Luke Church, Chris Nash, and Alan F. Blackwell. 2010. Liveness in Notation Use: From Music to Programming. In *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*. PPIG.

[4] Allen Cypher (Ed.). 1994. *Watch what I do: Programming by demonstration*. The MIT Press.

[5] A. A diSessa and H. Abelson. 1986. Boxer: a reconstructible computational medium. *Commun. ACM* 29, 9 (9 1986), 859–868.

[6] Christopher Michael Hancock. 2003. Real-Time Programming and the Big Ideas of Computational Literacy.

[7] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, 281–292.

[8] Peter Henderson and Mark Weiser. 1985. Continuous execution: the VisiProg environment. In *Proceedings of the 8th International Conference on Software Engineering (ICSE '85)*. IEEE Computer Society Press, London, England, 68–74.

[9] Joshua Horowitz. 2018. PANE: Programming with Visible Data. (2018). Presented at the Workshop on Live Programming (LIVE) 2018.

[10] Joshua Horowitz, Devamardeep Hayatpur, Haijun Xia, and Jeffrey Heer. 2025. Sculpin: Direct-Manipulation Transformation of JSON. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*. ACM, 1–15.

[11] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, 1–18.

[12] Joshua Horowitz and Jeffrey Heer. 2023. Live, Rich, and Composable: Qualities for Programming Beyond Static Text. (2023).

[13] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. https://doi.org/10.1207/s15327051hci0104_2. *Hum. Comput. Interact.* 1, 4 (1985), 311–338.

[14] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (feb 15 2023).

[15] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3363–3372.

[16] Alan Kay, Kim Rose, Dan Ingalls, Ted Kaehler, John Maloney, and Scott Wallace. 1997. Etoys SimStories. https://tinlizzie.org/VPRIPapers/hc_etoys_sim_1997.pdf.

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *International Conference on Electronic Publishing*.

[18] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 1–7.

[19] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3 (jan 2 2019), 1–32.

[20] W. Pugh and T. Teitelbaum. 1989. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*. ACM Press, 315–328.

[21] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (feb 1 2019).

[22] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (jul 23 2018).

[23] Randall B. Smith, John Maloney, and David Ungar. 1995. The Self-4.0 user interface. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. ACM, 47–60.

[24] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing* 1, 2 (6 1990), 127–139.

[25] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, San Francisco, California, 31–34.

[26] David Ungar and Randall B. Smith. 2013. The thing on the screen is supposed to be the actual thing. (2013). Presented at the Workshop on Live Programming (LIVE) 2013.

[27] Bret Victor. 2012. Inventing on Principle. (2012). Presented at the the Canadian University Software Engineering Conference (CUSEC).

[28] Bret Victor. 2012. Learnable Programming. http://worrydream.com/LearnableProgramming/.

[29] Viewpoints Research Institute, Inc. 2025. squeakland : home of squeak etoys. http://www.squeakland.org/. Accessed: 2025-12-14.

[30] Workshop on Live Programming. 2024. Live 2024 | Workshop on Live Programming. https://liveprog.org/.

[31] Parker Ziegler, Justin Lubin, and Sarah E. Chasins. 2025. Fast Direct Manipulation Programming with Patch-Reconciliation Correspondence. *Proceedings of the ACM on Programming Languages* 9 (jun 10 2025), 699–724.