The Blurry Boundaries Between Programming and Direct Use

JOSHUA HOROWITZ, University of Washington, USA

While a narrow conception of programming makes it possible to draw sharp lines between "programming" and "direct use", many of the efforts of the PL + HCI community call for expanded senses of programming that blur these lines. In this paper, we explore these murky boundaries from both sides, exploring situations where programming systems are used for direct use, and situations where direct-use systems take on the characteristic powers of programming.

1 Introduction

We often draw a distinction between the activity of *programming* and the activity of *directly using a computer*. But as we expand our view to include *end-user programming* [28], and as we look more carefully at programmatic mechanisms embedded in conventional software, we begin to see these lines blur. Alex edits a spreadsheet. Sam edits a CSS file to restyle their webpage. Jordan makes a button shape into a reusable component in the vector graphics editor Figma. Taylor moves files around with commands at a UNIX terminal. Who in this cast of characters is "programming"?

This paper explores this question, focused on complications that arise when *programming* and *direct use* overlap. Our goal is much less to present an answer to the question than it is to convince you that the answer is unclear and the question is worth talking about.

In fact, we believe that this question is of vital importance to the community of researchers at the intersection of PL and HCI. In this community, we often step away from the narrowest conception of programming to explore possibilities that tug at its boundaries. Programs might be edited as diagrams, rather than as text. Programmers might be end-users, rather than professional software engineers. And programming might even be done by working by demonstration on concrete data, rather than by editing a symbolic specification. While these situations may not match the superficial appearance of the most stereotyped modes of programming, the PL + HCI community embraces them, sensing that they match a deeper essence of programming.

But stretching the bounds of a conceptual space can get you into trouble. We have repeatedly encountered reviewers who object to classifying our work as "programming", because the work does not expose a textual format, or because it does not offer Turing-complete expressivity. We have heard similar reports from our colleagues. It appears that what does and does not qualify as "programming" is a site of active controversy. If we wish to continue exploring these further reaches without giving up the banner of "programming", we will need to clarify and argue for the things programming means to us.

The value of such clarification goes beyond armoring ourselves for academic turf battles. We need clarity to make progress. Some of the most exciting visions of programming, at least to the present authors, are those that intertwine programming and everyday computer use. In these visions, programming begins to take on the directness and visibility of direct use, and direct use begins to take on the expressiveness and power of programming. In imagining these hybrid systems, one begins to see aspects of programming already present in everyday computer use, in mechanisms like multi-cursor editing in text editors and component systems in graphics editors. Perhaps these aspects could be cultivated and extended. But to think clearly about these promising possibilities, one needs to make sense of a radically expanded notion of "programming".

This paper aims to chart out some of these blurry boundaries. In §2, we examine the relationship between programming and "the authoring of dynamic artifacts", identifying difficulties in this definition. In §3, we investigate the possibility that programming may describe certain kinds of

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

It feels like programming …

- … when you create dynamic artifacts, especially when the dynamics are bespoke and unpredictable.
- … when you can define reusable components.
- … when processes have persistent, editable representations.
- … when you can act on many objects at once, and when you have powerful tools for specifying what objects to act on.
- … when you have a smooth pathway from concrete action to parameterized action.

Fig. 1. A table of "feelings of programming", gathered from throughout the paper.

means, rather than ends, and we find hints of those means across many direct-use environments. We conclude, in §4, with a short discussion of the promise and peril of thinking of programming as a disciplinary lens. At no point will we claim to know what programming actually is. We will, however, pick up a few "feelings" along the way, previewed in Figure 1.

Throughout this paper, we adopt a predominantly technical perspective on the question "what is programming?" – a question that is, of course, entangled with significant historical and sociological factors. (Indeed, we have encountered these factors ourselves during review cycles – there is clearly a lot of boundary-work [15] happening here.) We hope that our technically-minded inquiry may be helpful fodder for future work on these topics, and direct interested reader to works such as Petricek [29], Lonati et al. [24], Ko [22], and Bullynck & De Mol [5] as entry points.

2 Programming as Ends

So then: what is programming?

In the interest of time, we will skip entirely past the narrowest conceptions of programming – those that demand Turing-completeness, a "general-purpose" range of applications, or text-centeredness. Instead, we will begin with a much more enlightened definition from Ko et al. [23]'s survey of end-user software engineering. They start by rejecting, as we do, definitions narrowly constraining the language in which programs are written, "requiring, for example, that the notation be Turing complete, and able to specify sequence, conditional logic and iteration". They proceed:

> [W]e define a program as "a collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities." Note that the variability of input values requires that the program has the ability to execute on future values, which is one way it is different from simply doing a computation once manually. This definition captures general purpose languages in wide use, such as Java and C, but also notations as simple as VCR programs, written to record a particular show when the time of day (input) satisfies the specified constraint, and combinations of HTML and CSS, which are interpreted to produce a specific visual rendering of shapes and text. [23]

We take the core of this definition to be the criterion that programmed artifacts "take variable inputs", and thereby have "the ability to execute on future values, [rather than] simply doing a computation once manually". We will call such artifacts "dynamic artifacts", in contrast to "static artifacts" like text documents and images. Note that this criterion is about the ends of programming, not the means. It examines what comes out of the process of programming, rather than characteristics of the process itself.

Artifacts can be dynamic in different ways. Perhaps the most obvious is that they can be dynamic by being *interactive*, responding in the moment to actions taken by a user. They can also be dynamic by responding to non-user data streams, such as real-time weather data. Finally, artifacts can be dynamic by being explicitly abstract and incomplete, like functions provided by a software library.

By defining programming in terms of its *ends* – what it *produces* – this definition adopts a stance of impartiality towards the means by which a system does this. This keeps the door open to diverse and radical approaches. For instance, it's easy to say, with this definition, that a visual programming system like Scratch [25] or a a programming-by-demonstration system like Wrangler [20] qualify as "programming", as they both clearly produce the same kind of dynamic artifacts that traditional languages do. We consider this a clear advantage of this definition.

What counts as dynamic? However, we immediately run into complications when trying to pin down which artifacts are and are not dynamic. A lot of this has to do with perspective: What qualifies as external "inputs" to an artifact, versus what is part of an artifact itself? For instance, Ko et al. provocatively claim VCR programming is programming, as a programmed VCR responds to the time of day in a dynamic way. However, this relationship between time of day and behavior seems exceptionally predictable and, one could say, static. We can tug on this example in the direction of other time-dependent "programs". It is said, colloquially, that one "programs" a drum machine. But a user working in a digital audio workstation like Ableton Live[1] is not usually thought of as a programmer, even as they arrange notes to be played in the future. For one thing, the final product of their work is typically an audio file, which seems like a static artifact akin to text documents and images. The same could be said of other time-dependent media, such as videos. It seems like a stretch to call someone splicing video clips together in Premiere [2] a "programmer" merely because they are arranging clips in time, to be played back later.

It is apparently unclear when unfolding in time makes an artifact dynamic. It is similarly unclear when affording interaction makes an artifact dynamic. Is a simple HTML file a dynamic artifact? You can scroll it. Perhaps the HTML has certain elements that remain "sticky" on scroll, so scrolling does not simply pan a fixed image. Perhaps the HTML contains "responsive" CSS, which makes it adapt to different screen widths. Perhaps we have a network of HTML pages linked together with hyperlinks. At what point do we cross the line from static to dynamic? Charts made with Google Sheets have interactive tooltips that appear on hover – does that make them dynamic artifacts, and make Google-Sheets chart-makers programmers? When are dynamics considered an authored part of a dynamic artifact, and when are they considered a part of how we interact with a static artifact?

Our suspicion: All else being equal, affording interaction does make an artifact seem more dynamic, and does make the act of creating it feel more like programming. But the more predictable these interactions are, and the less thought the creator has to put into them, the less we consider their activity to be "programming". An HTML page is scrollable, but an HTML author does not typically redefine the way this scrolling works. If they do, that begins to feel like programming.

> *It feels like programming …* when you create dynamic artifacts, especially when the dynamics are bespoke and unpredictable.

Dynamic means to static ends. A second, intertwined issue is that the "authoring dynamic artifacts" definition of programming would seem to exclude many things that feel an awful lot like programming. Picture a scientist using a Jupyter notebook to analyze a data set. They use every imaginable programming technique along the way, but at the end, they just write out a table of

[1] https://www.ableton.com/en/live/
[2] https://www.adobe.com/products/premiere.html

values and a chart, and forget about the notebook entirely. Is this programming? Tables and charts are undeniably static artifacts. It appears the scientist has used dynamic means to produce static ends – a common pattern, but one unaccounted for when defining programming in terms of ends. Or imagine a user writing a single-use Bash script at the command line to rearrange files on their computer in a complex way; a script that will not be saved nor run again. It certainly seems like they're programming, but they produce no artifact at all – just an effect on their file system.

While there is clearly a strong link between programming and "authoring dynamic artifacts", this criterion is neither necessary nor sufficient to characterize our intuitions. It seems difficult to get around this problem with a purely "external", ends-centric definition of programming.

3 Programming as Means

If an ends-centric conception doesn't effectively capture what programming is about, we should turn our attention to means. Here, we must tread carefully, since investigating the internal mechanisms of systems puts us at risk of reproducing narrow ideas about what these mechanisms should look like. As a strategy to keep our minds open, we will begin by looking for programmatic means not in situations conventionally described as programming (like the data-analyzing scientist), but in situations arising in direct-use computing.

Reusable components. What aspects of an activity make it feel like programming? Our first guide comes directly from our discussion of ends: When does a computer user construct, as part of their activity, dynamic artifacts as *intermediate tools*? That is, when do they define "specifications that may take variable inputs" as part of a larger workflow?

In direct-use contexts, such specifications often appear as reusable templates. For instance, most word processors (such as Google Docs or Microsoft Word) provide a system of labeled "styles", such as "Heading 1" and "Subtitle", which can be applied to text, setting attributes like font, size, and weight. When a user sets a style's attributes, they are performing an abstract act which may have no immediate effect on the document. Rather, they are setting up a tool for future use, in a way akin to a programmer defining a function. They are specifying a dynamic artifact.

One can imagine a primitive word processor implementing styles as imperative "macros", where applying a style to text simply applies the style's attributes in the moment. But modern word processors actually link the text to the style *symbolically*, meaning that changes to a style's attributes automatically update all previous uses of the style in the document. This feels, in some ways, even more programmatic than macro styles, as it embeds symbolic structures persistently into a document in a way we will later term "non-destructive".

Template systems in direct-use software grow in complexity from here. For instance, the vector graphics editor Figma features a "component" system in which a main component can be duplicated to form instances with a persistent link. Changes to the main component propagate to instances, while certain aspects of instances (like label text and color choices) can override the main component. Further levels of abstraction have been added to Figma's component system to meet the needs of graphic designers, such as the ability to define (multiple dimensions of) variants and the ability to parameterize instances with "properties".[3]

> *It feels like programming …* when you can define reusable components.

Persistent representations of process. Persistently linked, reusable templates build on an even more fundamental capability: "non-destructive" (aka "non-linear") editing. Traditionally, edits to a

[3] https://www.figma.com/blog/a-tale-of-two-parameter-architectures/

raster image in a program like Photoshop[4] simply edit the underlying pixels, and leave no symbolic trace. If a user wants to go back and, say, change the parameters of a filter they applied, their only option may be to go back in an undo history to before the filter was applied, apply the filter with new parameters, and redo their successive work on top of it. Fortunately, Photoshop offers more sophisticated structures, like adjustment layers, that make the application of a filter a persistent part of an image's structure, subject to later revision. Compare this to traditional programming, which specifies every step of a process in a persistent, symbolic form, allowing later editing. Traditional programs are the gold standard of non-destructive editing, and adding non-destructive features to direct-use systems brings them closer to a programmatic feel.

In the academic literature, *instrumental interaction* has been a prominent champion of persistently representing processes [2, 3]. One of the primary principles of instrumental interaction is *reification*: "turning concepts into objects". In practice, this often means reifying transient workflows into persistent parts of documents. For instance, StickyLines [9] reifies vector-graphics alignment and distribution tasks as first-class, persistent guidelines that shapes can be attached to. In our view, instrumental interaction pushes direct-manipulation interfaces closer to programmatic power. Beaudouin-Lafon & Mackay [3] write: "Turning commands into objects provides potentially infinite regression. Since instruments are objects, they can be operated upon by (meta)-instruments, which are themselves objects, etc." This potential for unlimited composability and abstraction is characteristic of programming systems.

Beaudouin-Lafon & Mackay [3] call out hierarchical "group" structures as a prime example of reification – by reifying a collection of objects into a new object called a "group", tree structures become possible. This possibility is exploited and extended by Cuttle[5], a vector graphics editor for making laser-cutter designs. Cuttle allows users to apply "live modifiers" to groups, which transform the group's contents with operations like repetition, boolean operations, and geometric transformations. These modifiers live persistently in the user's drawing. It is striking that, since these modifiers live in a tree-shaped hierarchy, they can in fact be combined in an open-ended way much like functions in a programming language.

> *It feels like programming …* when processes have persistent, editable representations.

Selections and parallel action. One hallmark of computer programming is the use of abstraction to specify repetition using constructs like loops or queries. Direct-manipulation systems have often been faulted for their limited ability to express repetition. Frohlich [13] lists "Performing repetitive actions" in his list of "tasks which are difficult to do by [direct] manipulation". Naturally, some parallel action is pervasive in conventional direct-manipulation interfaces – *e.g.* selecting a set of shapes in a vector-graphics editor and changing fill colors in a bulk operation. But direct-use systems have often been limited, both in their ability to express targets of parallel action and their ability to express actions themselves. We see hints of programmatic structure in ways that direct-use systems stretch these limitations.

For example, take multi-cursor editing in text editors. Powerful editors like Emacs have long had the ability to repeat actions through explicit macros or scripting. Most editors also have specialized parallel-action commands like search-and-replace. But more recently, it has become common for editors to offer the ability to operate on multiple cursors simultaneously.[6] While search-and-replace

[4] https://helpx.adobe.com/photoshop/using/nondestructive-editing.html

[5] https://cuttle.xyz/. Cuttle has a number of other interesting programmatic features, including reusable components, explicit parameterization, and scriptability.

[6] This feature appears to have been invented by Miller & Myers [27] and largely popularized by Sublime Text (https://www.sublimetext.com/).

bundles into a single command (1) finding a set of selections and (2) overwriting each selection with a replacement, multi-cursor editing reifies the intermediate set of selections into an interactive object that can be acted upon with the full range of selection-modification and text-modification actions available in the text editor. This makes search-and-replace actions more concrete and visible, while opening up a vastly extended range of possible uses. For instance, multiple cursors make it easy to transform all instances of function calls like `multiply(vec, 10)` into method calls like `vec.multiply(10)`, using cursor movement commands and copy-paste, rather than complex, symbolic regular-expressions.[7]

The usefulness of multi-cursor editing replies upon the semantically expressive range of commands made available by text editors. In order to effect the transform described above, the user must move the cursor from the selections matching `multiply(` (which they obtained via a search) to the first parameter of the call, like `vec`. Once the cursor is after the open-parenthesis, they can select this first parameter with a single "word-wise" cursor movement command (conventionally "shift-alt-right"). Each cursor may run into a different identifier in this position (like `vec2` or `circleVec`), but word-wise selection will pick all of these up correctly. In single-cursor editing, access to word-wise cursor movement is a helpful speed-up, but never essential – a user can always just press "left" or "right" until they get where they want to go. But in the presence of multi-cursor editing, higher-level operations like word-wise cursor movement become indispensable. They are what allow users to express intent in a way that can generalize across multiple selections, even as the details of the selections vary. For parallel action to be expressive and useful in direct-use systems, these systems need commands that generalize.

Another prominent limitation of parallel action in direct-use systems is the difficulty of specifying what objects to act on. For instance, Buxton [6] discusses how despite making it easy to specify a single object by demonstration, direct-manipulation systems typically fail "to adequately support… the **descriptive** specification of operands" (emphasis added), giving as an example a user wishing to replace all instances of one structure in a circuit editor with a different structure. Direct-use systems have made progress on this front. As a humble example, when a set of objects are selected in Figma, a side-panel shows a palette of colors appearing among the selected objects. From this panel, colors can be directly replaced (a search-and-replace-like action), or, better yet, the selection can be narrowed to objects with a given color. The ability to "query" a selection like this is extended by Collection Objects [34], which makes selections persistent and filterable by arbitrary attributes, and by Sculpin [19], which centers the "sculpting" of multi-cursor-like selections through steps of generalization, navigation, and filtering.

> *It feels like programming …* when you can act on many objects at once, and when you have powerful tools for specifying what objects to act on.

Non-programming programming and "abstractability". Throughout this work, we have been inspired by Hoff's *Always Already Programming* [18]. Hoff complicates, as we do, the distinction between using and programming computers: "When a programmer is writing javascript, they are using prewritten, packaged functions and variables in order to carry out the actions they want their code to do. In this way, the programmer is also the user. Why is using pre-made scripts seen so differently than using buttons that fire pre-made scripts?". While we tend, like Hoff, to be programming maximalists, seeking programmatic structures in unlikely places, we can also

[7] Forest [33] extends multi-cursor editing to operations directly on abstract syntax trees, making parallel operations like these even more reliable.

interpret Hoff's question in the opposite direction. When a textual program only replicates actions that could be done in a direct-use interface, should it count as programming?

Victor's *Learnable Programming* [31] touches on this question.[8] In this essay, Victor begins by taking a canvas drawing API and building a bidirectional editor for it, in which calls to `fill()` and `rect()` are controlled by direct manipulations on the canvas. He reflects: "With this interface, is this even 'programming'? No, not really." But, writes Victor, neither is using the canvas drawing API to draw shapes with fixed cartesian coordinates. "It's merely a very cumbersome form of illustration. It becomes genuine programming when the code is abstracted – when arguments are variable, when a block of code can do different things at different times." Here, Victor closely echoes Ko et al. [23]'s definition we discussed earlier – programming is about the creation and use of dynamic artifacts, not merely the use of formal textual syntaxes.

If writing lines of code like `triangle(80,60, 80,20, 140,60)` isn't programming, what other things might not be? For instance, the Bash language used at the Unix command line is certainly a programming language. But much of its use is unabstract in the same way this triangle is. If one types `mv old.mp4 new.mp4` to move a file, this is no more abstract than dragging the file in a file browser. If one types `ffmpeg -i old.mp4 -vcodec libx264 -crf 18 new.mp4` to re-encode a video, this is no more abstract than loading the video in a GUI like HandBrake, specifying parameters in a form, and pressing "Start".

But there is, in fact, an important difference. Even though these actions are not abstract, their representation as code in larger programming systems makes the pathway into abstraction much more clear. Each parameter to `triangle` or `mv` or `ffmpeg` provided as a constant literal can quickly be replaced with a parameter, or the call itself can easily be inserted into a loop or a conditional. We may say that although the call is not *abstract*, it is *abstractable*. This is not at all the case in conventional user interfaces. Dragging a corner of a triangle in a vector-graphics editor does not reveal a "slot" for parameterization. And actions in direct-use systems are typically immediately forgotten, or at most placed in an undo history. They are seldom reified as objects that can be looped or made a part of a larger, parametric process.

> *It feels like programming …* when you have a smooth pathway from concrete action to parameterized action.

Bridging the gap. What would it mean for dragging the corner of a triangle to present a slot for parameterization? In other words, what infrastructures might give direct-use systems a gentle pathway into programmability and abstraction?

One set of possibilities lies in *bidirectional* programming systems, exemplified by Sketch-n-Sketch [8, 16, 17]. These systems interpret direct manipulations of media as edit actions on an underlying (generally textual) program that generates this media. Bidirectional systems vary in the variety of edits that can be accomplished through direct manipulation. For instance, the classroom-oriented bidirectional systems presented in Do et al. [11] and Mcnutt et al. [26] do not provide ways of manipulating programmatic structure with direct manipulation, merely creating and modifying concrete shapes. This accomplishes the important goal of creating a smooth pathway from concrete action to parameterized action – shapes created through direct manipulation can then be adapted into structures in code. But it leaves the "direct manipulation" side of the system quite limited.

At the other end of the spectrum lie systems like Sketch-n-Sketch, which aim to make it possible to edit arbitrary code written in general-purpose languages using operations on output. If

[8] Keer draws the connection to *Learnable Programming* in her insightful response to Hoff [21].

successful, this approach will be tremendously expressive, and gain many advantages from its compatibility with existing programming workflows. However, such systems may be held back by their dependence on existing textual-language structures. The mechanisms direct-use systems develop to gain programmatic power rarely map one-to-one with conventional programming-language idioms. Rather, they are designed around the ontologies and interactions found in the direct-use systems hosting them. For example, while Figma components can receive explicit function-like parameters, instances of components can also override parts of the main component in an ad-hoc fashion. Other direct-manipulation programming systems like Apparatus[9] and the "managed copy and paste" prototype from Edwards & Petricek [12] stretch this idea even further. A system like Sketch-n-Sketch, centered around traditional programming-language constructs, has less latitude to explore mechanisms like these.

Direct-manipulation programming systems which do not insist on maintaining bidirectional compatibility with conventional textual code can often be classified as "programming by demonstration" [10]. For example, Victor's *Drawing Dynamic Visualizations* prototype (DDV) [32] presents a vector-graphics interface where actions on shapes are recorded into a timeline. By adding loops over data to this timeline, and linking parameters of actions to data attributes, Victor is able to use his tool to build bespoke dynamic data visualizations. DDV recruits many features of direct-use graphics editors into programmatic use. For instance, relationships between objects can be established by "snapping" during a drag. We can contrast this approach with that of StickyLines [9] – whereas StickyLines reifies a snapping interaction into a novel object that lives persistently *inside* a drawing, a snap operation in DDV is instead persisted *outside* a drawing, as an imperative step in the timeline. This "macro"-style approach is also adopted by Wrangler [20] and Sculpin [19]. It has advantages: the Sculpin authors motivate it by appeal to referential transparency [4], as well as the ease of generalizing this approach to new media by avoiding medium-specific programmatic extensions. However, the "macro"-style approach has weaknesses – Bakke & Karger [1] critique "algebraic" systems for the difficulty they present to editing programs after they are first written.

4 Programming as a Lens

Perhaps what makes a given situation "programming" is that the programming-languages (PL) community can turn their eye to it and think of it as programming. That is, we may want to view "programming" not as an inherent quality of an activity, but as a lens we can apply to any activity. For instance, how might we think about a Photoshop document as a program? Perhaps we might see the arrangement of layers and filters as the "syntax" of this program, and see compositing the document into a raster image as a "semantics". From this perspective, we might ask what sort of static analysis is possible – are there properties of a document we might detect without actually running it? We might also call upon traditional patterns and techniques of programming languages. Can you save layers into variables and refer to them elsewhere? Can you combine arrangements of layers into parameterized procedures?

Applying the lens of PL to unconventional settings can be extraordinarily generative. This is, essentially, half of the call of Chasins et al. [7]: that those with PL expertise can apply it in fruitful ways to problems across the range of HCI. Shaw [30] makes a similar case for domain-specific end-user programming: "Many [domain-specific languages] have been designed ad hoc, and they would benefit from the formal rigor that's brought to the design of general-purposes languages".

However, elevating the disciplinary lens of PL carries some dangers. The discipline of PL does not examine programming in a value-neutral way. Rather, it carries path-dependent (and not entirely disinterested) assumptions and ways of approaching problems. For instance, the PL community

[9] https://aprt.us

tends to celebrate sophisticated static type systems. This may come from the mathematical appeal of types, and from the needs of large software-engineering organizations. Yet static types may or may not be appropriate as we blur the boundaries of programming into other spaces, like end-user use. Shaw makes this argument in her paper, listing "myths" that PL practice implicitly embeds in their work, like the "Mathematical Tractability" myth that "[s]oundness of programming languages is essential" and the "Specifications myth" that "formal specifications are [...] essential" [30]. PL's disciplinary lens is a powerful tool, but we must take care that it does not unduly circumscribe the possibilities we explore.[10]

Acknowledgement

Many of the perspectives offered in this paper arose during collaboration with Devamardeep Hayatpur. We also thank participants in the Pacific Programming Interfaces Confab for helpful feedback on a draft.

References

[1] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1377–1392.

[2] Michel Beaudouin-Lafon. 2000. Instrumental interaction. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 446–453.

[3] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, polymorphism and reuse. In *Proceedings of the working conference on Advanced visual interfaces*. ACM, 102–109.

[4] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (12 2011), 2301–2309.

[5] Maarten Bullynck and Liesbeth De Mol. 2024. The Myth of the Coder. *Commun. ACM* 67, 9 (aug 26 2024), 20–23.

[6] Bill Buxton. 1993. HCI and the inadequacies of direct manipulation systems. *ACM SIGCHI Bulletin* 25, 1 (1 1993), 21–22.

[7] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI. *Commun. ACM* 64, 8 (jul 26 2021), 98–106.

[8] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 341–354.

[9] Marianela Ciolfi Felice, Nolwenn Maudet, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2016. Beyond Snapping. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 133–144.

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

[11] Quan Do, Kiersten Campbell, Emmie Hine, Dzung Pham, Alex Taylor, Iris Howley, and Daniel W. Barowy. 2019. Evaluating ProDirect manipulation in hour of code. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. ACM, 25–35.

[12] Jonathan Edwards and Tomas Petricek. 2022. Interaction vs. Abstraction: Managed Copy and Paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. ACM, 11–19.

[13] David M. Frohlich. 1993. The history and future of direct manipulation. http://www.tandfonline.com/doi/abs/10.1080/01449299308924396. *Behaviour Information Technology* 12, 6 (11 1993), 315–329. [Online; accessed 2025-12-15].

[14] Richard P. Gabriel. 2012. The structure of a programming language revolution. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 195–214.

[15] Thomas F. Gieryn. 1983. Boundary-Work and the Demarcation of Science from Non-Science: Strains and Interests in Professional Ideologies of Scientists. *American Sociological Review* 48, 6 (12 1983), 781.

[16] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 379–390.

[10] Gabriel writes about the limitations of disciplinary lenses in his essay on "The Structure of a Programming Language Revolution" [14], describing how shifts in the PL community during the 1990s made it more difficult to engage with systems-oriented work, as opposed to language-oriented work.

[17] 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.
[18] Melanie Hoff. 2020. Always Already Programming. https://gist.github.com/melaniehoff/95ca90df7ca47761dc3d3d58fead22d4.
[19] 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.
[20] 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.
[21] Lucy Keer. 2021. Always Already Programming. https://lucykeer.com/notebucket/always-already-programming/.
[22] Amy J. Ko. 2016. What is a programming language, really?. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 32–33.
[23] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *Comput. Surveys* 43, 3 (4 2011), 1–44.
[24] Violetta Lonati, Andrej Brodnik, Tim Bell, Andrew Paul Csizmadia, Liesbeth De Mol, Henry Hickman, Therese Keane, Claudio Mirolo, and Mattia Monga. 2022. What We Talk About When We Talk About Programs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*. ACM, 117–164.
[25] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (11 2010), 1–15.
[26] Andrew M Mcnutt, Anton Outkine, and Ravi Chugh. 2023. A Study of Editor Features in a Creative Coding Classroom. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, 1–15.
[27] Robert C. Miller and Brad A. Myers. 2001. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 161–174.
[28] Bonnie A Nardi. 1993. *A small matter of programming*. MIT Press, London, England.
[29] Tomas Petricek. 2019. Cultures of Programming: Understanding the History of Programming through Controversies and Technical Artifacts. (2019). Unpublished draft.
[30] Mary Shaw. 2020. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages* 4 (jun 14 2020), 1–44.
[31] Bret Victor. 2012. Learnable Programming. http://worrydream.com/LearnableProgramming/.
[32] Bret Victor. 2013. Drawing Dynamic Visualizations. Presented at the Stanford HCI seminar on February 1, 2013.
[33] Philippe Voinov, Manuel Rigger, and Zhendong Su. 2022. Forest: Structural Code Editing with Multiple Cursors. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 137–152.
[34] Haijun Xia, Bruno Araujo, and Daniel Wigdor. 2017. Collection Objects. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 5592–5604.