Programming Ecology

ERIC RAWN, University of California, Berkeley, USA
JUSTIN LUBIN, University of California, Berkeley, USA
CLEMENS KLOCKMOSE, University of Aarhus, Denmark
S.E. CHASINS, University of California, Berkeley, USA

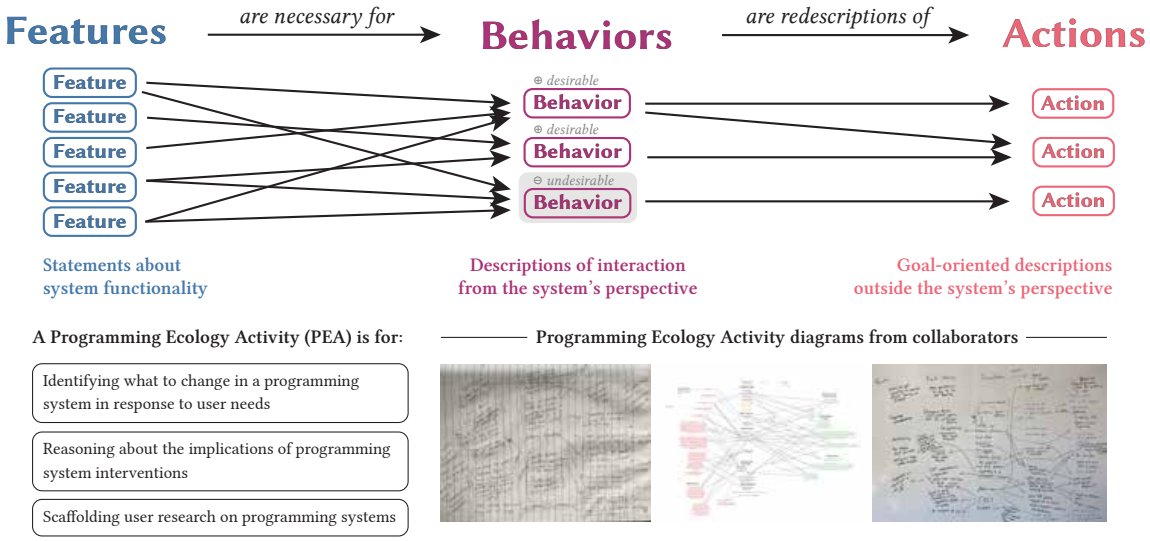


Fig. 1. A *Programming Ecology Activity* (PEA) is a design tool for reasoning about how the features of a programming system collectively relate to the diverse ways programmers are observed to use a system. A PEA involves understanding a system and the user research on that system through three key terms: **features**, **behaviors**, and **actions**. By understanding the relationships between these terms, a PEA supports programming systems designers in identifying features to change based on user needs, reasoning about the implications of system interventions, and supporting the design of future user research on the system. We have developed the ideas behind Programming Ecology Activities over the course of four case studies with HCI programming systems researchers.

Programming languages and tools are built to be intentionally open-ended—ready to be used for many different tasks and in creative and hard-to-anticipate ways. Like other design tasks, the programming systems design process revolves around supporting users in achieving desirable behaviors and putting undesirable behaviors out of reach. However, this process is hard for programming systems designers. Because tools are intentionally compositional, any given user behavior may depend on multiple interacting system features. Because tools intentionally support a wide range of user activities, the same programming systems features make many user

Authors' Contact Information: Eric Rawn, erawn@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Justin Lubin, justinlubin@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Clemens Klockmose, clemens@cs.au.dk, University of Aarhus, Aarhus, Denmark; S.E. Chasins, chasins@cs.berkeley.edu, University of California, Berkeley, Berkeley, CA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

behaviors possible. We propose a design activity, which we call the Programming Ecology Activity, developed to support the designers of programming languages, programming environments, and other programming tools. We describe what we have learned about Programming Ecology Activities so far via case studies of four HCI–PL research teams designing a range of programming systems.

CCS Concepts: • **Human-centered computing → HCI theory, concepts and models**; • **Software and its engineering → Software notations and tools**.

ACM Reference Format:
Eric Rawn, Justin Lubin, Clemens Klockmose, and S.E. Chasins. 2018. Programming Ecology. In *Proceedings of (Preprint).* ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Many domains of work demand continual improvisation, discovery, and open-ended creation from their practitioners [53, 54, 58]. Practitioners who have such ever-evolving work often turn to *programming systems* to address their computational needs. We use the term "programming system" [23] holistically, encompassing programming languages (e.g., Python [48], R [49]), textual programming environments (*e.g. VSCode [38], Eclipse [59]*, literate programming environments like Jupyter Notebooks [13], RStudio, and Observable [45]), graphical programming environments (*e.g.,* Scratch [50], Sketch-n-Sketch [19, 33]), domain-specific tools that help users specify novel procedural computation (*e.g.,* Houdini, Tableau, Excel), recent LLM-backed programming tools (e.g., Claude Code, Cursor), and other programming tools that fall in none of these categories (e.g., Tidal, TouchDesigner, Max/MSP).

Programming systems are well-suited to open-ended, discovery-oriented work in large part because each system supports a massive diversity of interactions despite its fixed set of system features. The resultant flexibility and user control is excellent for users but difficult for designers. Open-ended interactions make it daunting to design system interventions and study the use of programming systems:

(1) **User frustrations can be symptoms of underlying system features that are difficult to identify.** For example, confusions around global variables have been a known, persistent, and severe difficulty for computational notebook users for years [7, 22, 26]. Yet it is difficult to identify precisely what parts of the system cause this issue. Is it due to a feature of the Python language, the IPython environment, the JupyterLab interface, or some interaction between the three? Tracing the path from a known user issue to its source in the programming system is not straightforward.

(2) **Seemingly small system changes can have far-reaching and unintended consequences.** For example, adding mutable references to a programming language can invalidate the safety of subtyping [47, §15], parametric polymorphism [12], type inference [60], effect inference [32], compiler optimizations [1, §9], and parallelism [1, §11]. This has had real-world consequences: Java's array covariance subtyping is famously unsound without slow runtime checks in the presence of mutability [47, §15.5.2], and it took years of research to figure out how to ergonomically combine mutable references with Hindley-Milner style typing [64]. A seemingly small change, designed to facilitate a known set of desirable behaviors, may also facilitate many undesirable behaviors.

(3) **User interactions can rely on a conjunction of system features working together in complex ways.** Programming system designers must reason about how any intervention interacts with all other parts of the system. For example, Rust is sometimes chided for slow compile times. One factor contributing to this is its aggressive monomorphization of code. But zero-cost abstractions like asynchronous functions depend on monomorphization for performance. They also depend on the lifetime system for memory safety, and lifetime

> resolution can be slow because it uses a Datalog-like logic engine at compile time—and lifetimes are exactly how Rust gets around needing a garbage collector in the first place. Compilation could instead have increased parallelism with small compilation units, but the trait system forbids trait implementations for structs outside the current compilation unit, and removing this rule would remove guarantees about trait coherence. The designs of these features are deeply interdependent, so choices about their designs cannot be made in isolation.

In each case, programming systems designers face a difficulty in reasoning about the relationship between a programming system's features and how users interact with the system. User research techniques (*e.g.,* contextual inquiry [21] and participatory design [15, 56, 63]) can help researchers identify how programmers use a system and the difficulties they experience, but effective programming system interventions seem to require more: a holistic understanding of a system as a network of features that together relate to the broad range of outcomes that users care about. How might design tools better aid programming systems designers in forming this understanding and building better systems?

As Human-Computer Interaction and Programming Languages researchers, building this understanding seems to require occupying a perspective from each discipline:

From a Programming Languages perspective, we need to reason about how the features of a programming system (and changes to those features) enable and constrain entire classes of what a system makes possible. In the limit, designers need an understanding of how their system functions in *any* possible circumstance. In the case studies described in this paper, many of our collaborators were building programming systems that guarantee that *any* program generated with the system will have certain properties (*e.g.,* validity). Reasoning about *all* programs that are possible and impossible to express enables designers to build robust systems, even when the programs users will express cannot be anticipated in advance. More broadly, thinking about what a programming system's features make *possible* and *impossible* enables designers to reason about a system's capabilities generally, helping to avoid unintended consequences and make more informed design decisions, even when programmers use the system to express novel computational behavior.

But knowing what a system makes possible tells us nothing about how programmers *actually use* a system or what they value about it. As Human-Computer Interaction researchers, we wanted system design decisions to be clearly motivated by close observation of and engagement with users. However, connecting those observations to particular system features is hard. To draw an example from one of our own case studies: many HCI researchers have identified through qualitative research that programmers who use Jupyter Notebooks often get overwhelmed with global variables. How should we go about identifying what features of the system are at fault given this user research? How should we reason about whether to intervene within the JupyterLab interface, the IPython kernel, or the Python language, and how can we intervene without causing other issues for the diverse ways programmers use the system? How might we build sufficient evidence that a change, which may affect many other parts of the system, is warranted?

If designers have built an understanding of how features of the system make certain interactions possible or impossible, how might they use that understanding to frame opportunities for design or to reason about potential effects on users when those system features change? While a design tool cannot reduce this complexity, it might help designers navigate it more easily.

In this paper we offer a set of design concepts for bringing these two perspectives on programming systems together. For designers coming from a mostly system-centric process, these concepts may support them in grounding their design in user research. For designers coming from a mostly user-centric process, these concepts may support

them in connecting their design to the interactions that a system makes possible and impossible. We focus on three design concepts that help us talk about how a user interacts with a programming system: **features**, statements about systems and their functionality; **behaviors**, system-centric descriptions of user interaction; and **actions**, user-centric descriptions of user interaction.

We introduce the **Programming Ecology Activity** (PEA), an exercise for programming systems designers. This exercise guides designers through explicitly mapping the connections between **features**, **behaviors**, and **actions** to reason about their programming systems and how users interact with them. We draw from an *ecological* perspective in HCI [3, 34] to see programming systems as a myriad of features that all work together to support a wide range of interactions. Rather than try to isolate any particular feature or user interaction in a design process, a Programming Ecology Activity aims to help designers reason about how the many features and interactions their system supports are woven together, understanding each in relationship to the broader system in which it participates.

We developed the ideas behind Programming Ecology through collaboration on four PEAs. Our PEAs covered a range of research projects at the intersection of Human-Computer Interaction and Programming Languages, each of which was developing either an intervention into an existing programming system or a novel programming system. The programming systems at the core of these PEAs ranged from standard computational notebooks to a new environment combining LLM-backed programming and visual programming. For each project we performed a PEA together with the research team to help answer whatever design questions were pressing. We present these experiences as case studies below to explore the value and flexibility of the exercise and its associated design concepts.

We have found a *Programming Ecology* perspective most helpful at three different moments in the design process:

(1) **Understanding the Use of Existing Systems:** What parts of a system are important for the outcomes users care about, and which are not? How can we identify opportunities for design?
(2) **Understanding the Consequences of Interventions:** How can we reason about the impact of system design changes on the diverse ways programmers interact with a system? Can we anticipate potentially far-reaching consequences during the early stages of system design?
(3) **Guiding User Research:** How can we frame user study questions that enable us to more deeply explore our theories of how our system design relates to user outcomes? How can our theories of our systems better resource our qualitative research on their use? What should we observe or measure in a programming study to give us insight on the theories and constructs we care about?

While these questions are relevant to a variety of systems design, in this paper we explore how programming systems might require new ways of answering them.

We present the following contributions:

(1) The **Programming Ecology Activity** (PEA), a design exercise for understanding and reasoning about programming systems and how they are used.
(2) A report of four case studies that put the PEA into practice to identify opportunities for design, assess programming systems interventions, and guide user research in programming systems.
(3) Reflections on how a Programming Ecologies perspective in HCI and PL can scaffold theory-building, support longitudinal and participatory user research, and support designers in making system interventions with more consequences with more confidence.

2 Related Work

2.1 HCI Techniques for Designing Programming Systems and Other Software

The intersection of programming languages and human-computer interaction is a blossoming area of research that is starting to see different perspectives on techniques for designing programming systems [6]. The *Cognitive Dimensions of Notation* [14] is an early and influential descriptive framework for reasoning about the different dimensions along which a programming system may vary, such as the degree to which a program requires premature commitment or makes visible underlying dependencies. As a descriptive framework, the Cognitive Dimensions of Notation does not prescribe a technique for designing system. However, building on the Cognitive Dimensions of Notation and earlier empirical studies of programming, researchers have advanced a pluralistic approach of applying user-centered design techniques such as contextual inquiry [21] and Wizard of Oz prototyping [9, 25] to programming system design [40, 41], including developing creative new approaches to feed into the user-centered design paradigm such as natural programming elicitation [46]. Recently, Coblenz et al. [8] integrate many of these techniques into a cohesive user-centered design process called the *Programming Language Iterative Evaluation and Refinement System* (PLIERS). As part of user-centered design process, these methods all largely target improving the usability of a design rather than creating and understanding the implications of possible new designs. As we discuss further in Section 8.3, such methods are complementary with our programming ecology perspective; where we are interested in whether features make certain behaviors possible or impossible (and the network of implications of these connections), user-centered design methods aim to improve the usability of a system.

In parallel, researchers have pushed forward a methodology of randomized controlled trials with quantitative metrics to assess the impact of, for example, keyword names [57], static types [18], and anonymous functions [61]; while these studies provide precise information for individual features of a programming system, they do not provide support for reasoning about what features to ablate nor for the the design of a programming language as a whole. To help execute such experiments and other controlled in-lab experiments, researchers have provided a variety of guidelines [10, 28].

2.2 Ecological Approaches in HCI Design

While many distinct perspectives in HCI research have used the term "ecology" [34], many share a focus on studying *networks of relationships*: between technologies that are used together [24, 36], between members of a workplace [27, 42] or community [20, 29], or the activity of individuals [3, 4, 30].

From a *Programming Ecology* perspective, a programming system is already a complex, interconnected network of computational artifacts (**features**) which programmers utilize together to achieve their goals (**actions**). Although the tools of a PEA can also be used to analyze multiple programming systems used together (which our case studies did not explore), we see a programming system as already a rich tangle of computational components which programmers collectively utilize.

To reason more effectively about the relationships between the numerous features of a programming system and the wide range of activity they support, we needed a conceptual frame from which we could make claims about what a systems' features make possible or impossible at all. For comparison, consider the Human-Artifact Model [3], which investigates the relationships of four aspects of an artifact-user coupling (motivational, instrumental, operational, and adaptive) that are considered from both perspectives: how artifacts are designed and how users orient themselves towards that artifact through their goals, experiences, and values. This distinction is mirrored in *Programming Ecology* with *Features* and *Actions*. In this way, we share an interest in a "dialectical" understanding of systems-in-use as always

involving a productive tension between designers' and users' perspective on what a system makes possible to do, how it is possible, and why the system behaves as it does. But where the Human-Artifact model is primarily concerned with the relationship between individual artifact-user couplings across different moments in time, *Programming Ecology* sees a single "artifact" as itself already an ecology of features supporting a wide ensemble of actions. Programming systems designers are often focused on supporting general classes of possible uses, and therefore connecting the designers' and users' perspectives requires reasoning about the relationship between what a system makes possible and how real users put a system into practice to their own ends. To do this in a way that retains a grounding in empirical observation, we introduce a mediating term between a system's **features** and the **actions** of a programmer: **behavior**, which is a redescription of **action** using only system-level terms. The concept of **behavior** enables us to make claims about what **features** are *necessary* for a given **behavior**. The concept of **behavior** gives programming systems designers a way to reason generally about how the same system **features** relate to many **actions** of a diverse group of users: by how a system's **features** make possible a wide range of **behaviors**.

3 Putting Programming Ecology Concepts to Work with Programming Ecology Activities

We have used Programming Ecology concepts in our work through a *Programming Ecology Activity* (PEA), a design exercise in which programming systems designers iteratively build a diagram that theorizes how the features of their system relate to how programmers utilize that system. We discuss the ways we've put PEAs into practice in Sections 5, 6, and 7, but in this section we first introduce the concepts required to read one of these diagrams and show how a programming systems designer can get started on one.

3.1 Key Concepts

At the most basic level, a Programming Ecology Activity comprises development of three different kinds of descriptions about a system and how programmers use that system—**actions**, **features**, and **behaviors** (Figure 1)—then drawing relationships between those items.

3.1.1 Actions: Goal-Oriented Descriptions Outside the System's Perspective. Sourced from direct observations of users programming, **actions** are descriptions of what users are accomplishing when they work in a system, and are often the colloquial ways both programmers and designers describe interaction (although they may use different language). Drawing from our case studies, examples of actions would include:

(1) "Comparing two versions of an analysis in a Jupyter Notebook"
(2) "Learning about an operator on a dataset by applying that operator on one's own data"
(3) "Reasoning locally about a program generation step"

Actions are often where a PEA begins because they are the descriptions most readily at hand once designers have done any kind of programming observation (*e.g.* participant observation [39], contextual inquiry [21], *etc.*). In each case, what the programmer is doing in the system is described in terms of what it means *for them*—"versions," "analysis," "learning," and "reasoning locally," are all terms that correspond to how a programmer is making meaning in their situation (even if the programmers themselves would not necessarily use those words verbatim), and do not correspond directly to something in the programming system.

3.1.2 Features: Statements about System Functionality. **Features** are descriptions of how a system works without reference to what users *do* with that system. Examples include, "all programs generated by the system are valid," "system

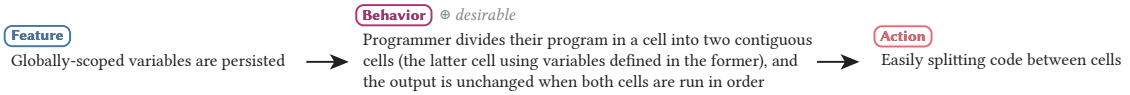


Fig. 2. A sample excerpt of a Programming Ecology Activity (PEA) diagram demonstrating one **feature**–**behavior**–**action** throughline.

shows the result of an operation at interactive speed," and "globally-scoped variables are persisted." While "feature" is a common term to describe an aspect of an interactive system, understanding the most useful details and granularity for describing a programming system's **features** can be difficult. As we discuss below, collaborators often came to understand their systems in new ways by splitting apart a single **feature** description into many distinct parts over the course of a PEA, critiquing and refining how they conceptualized their own systems.

3.1.3 Behaviors: Descriptions from the System's Perspective. **Behaviors** are descriptions of what users do in a system *using only system-level descriptions.* In contrast to **actions**, **behaviors** should only include terms that correspond to the programming system itself, and not descriptions of what users are accomplishing by using the system. While, of course, a system does not literally have a perspective, asking "What is happening in this interaction from the system's perspective?" was a helpful heuristic in our collaborations. To provide examples, the behaviors we generated[1] from the actions listed above (in Section 3.1.1) are, respectively:

(1) "Programmer defines two cells which read and write the same variables and execute them individually, along with any downstream cells which read the variables those cells write"
(2) "Programmer sees the result of an operator on the currently loaded data appear on the screen at an interactive speed"
(3) "Programmer selects between possible next steps while only being shown information about the current step"

As you might notice, **behaviors** can be much more detailed than **actions**—there is often quite a lot happening in a programming system even when users perform "simple" **actions**! In a PEA, **behaviors** serve as a mediating term between **features** and **actions** and sit at the boundary between describing a system and its general capabilities and describing users and particular observations of them.

3.1.4 The Relationship between Behaviors and Actions is Redescription. In a PEA diagram, we connect **behaviors** to the **actions** from which they are generated. In other words, **behaviors** are related to **actions** when they are both descriptions of the same user interaction. In a diagram, we could read an arrow pointing from a **behavior** to an **action** as "is a redescription of." Maintaining both a clear distinction between **behaviors** and **actions** while keeping related pairs linked helps designers continually confirm that **behaviors** remain faithful redescriptions while making visible the difference in what each description provides license for designers to claim.

3.1.5 The Relationship Between Features and Behaviors is Necessity. Once we have a **behavior** (a machine-centric description of an interaction), we can now ask, "what **features** of the system are *necessary* for this **behavior** to be possible?" These kinds of claims—about what **features** are *necessary* for certain **behaviors**—are the heart of a PEA. For example, consider the Jupyter Notebook **behavior** "programmer divides their program in a cell into two contiguous cells, the latter cell using variables defined in the former, and output is unchanged when both cells are run in order"

[1]We use the term *generate* to indicate our constructivist interpretation of PEAs (see Braun and Clarke [2]) and their complementary nature to existing approaches to qualitative user research in HCI.

(Figure 2). A necessary **feature** for this **behavior** could be phrased as "globally-scoped variables are persisted"—in other words, if globally-scoped variables *were not* persisted, then that **behavior** would be impossible—the program would error due to an undefined reference.

Crucially, we can make this impossibility claim without observing users because it does not rely on user observations as evidence—we know that, in every case, the program would throw an error. By contrast, a corresponding **action** for this behavior **behavior**—say, "easily splitting code between cells"—*does* rely on user observation: whether users can accomplish an action *easily* is an empirical claim. By generating a **behavior** from this **action**, we can identify **features** which are *necessary* for that **behavior** while making clear where our reasoning about the general capabilities of programming systems stops and our understanding of users begins: we could now hypothesize about the consequences of making this **behavior** impossible for the **action** we observed, which would raise a question only further user research could answer.

3.2 Getting Started With A Programming Ecology Activity

While a PEA is intended to be flexible to the needs of designers, in this section we present one common way we started our collaborations.

Step 1: Enumerate Actions from User Research. A PEA requires a Programming Ecology to discuss—at minimum, a real user using a realized system. In the collaborations we discuss below (See Section 4), collaborators started from a wide variety of places, but each had observations of how a programmer used their system of interest: existing user study literature (Notebook), participant observation and co-design (Bio), contextual inquiry (MapDM), workshops (TextProc), in-situ user studies (Notebook), and in-lab studies (MapDM). Each of these sources of data differs greatly in terms of what it reveals about programmers and what claims it supports researchers in making (and PEA practitioners should be aware of this), but for now the crucial point is that each provides a way to observe how a programmer uses a programming system. Begin by listing all of the **actions** identified from a chosen source of user observation, and categorize by how desirable those **actions** are (or hypothesized by designers to be) for users. For example, some **actions** in our case studies were clearly desirable (*e.g.* "smooth interactions across documents"), some clearly undesirable (*e.g.,* "variable confusion"), and others ambiguous (or ambiguous to researchers at the time).

Step 2: Generate Behaviors from Actions. Once **actions** have been enumerated, generate **behaviors** by asking, for each **action**, "what is happening from the system's perspective in this interaction?", drawing a line between a **behavior** and the **action** it was generated from. **Behaviors** should not include any terms that refer to how a programmer is interpreting their situation, their goals, or their reasons for interacting. Generating **behavior** descriptions requires a deep familiarity with a programming system, and we often found it helpful to recreate the observed **action** in the programming system ourselves during a PEA to scrutinize the details. Sometimes multiple **behaviors** can be generated from one **action** (*e.g.,* "being exploratory in a Jupyter Notebook" might refer to a cluster of related but distinct **behaviors**).

Step 3: Generate Features from Behaviors. Now that many **behaviors** have been generated, ask "what **features** in this system are *necessary* for these **behaviors**?" Experiment with formulating **features** that are necessary for multiple **behaviors**, especially those that are necessary for desirable *and* undesirable **behaviors**. These **feature** descriptions change over the PEA as the designers' understanding of the system evolves. Although feature descriptions should be

Table 1. We collaborated with researchers on these four Programming Systems projects with a Programming Ecology Activity. Each is referred to by an anonymous project label.

Project Label	Project Description
MapDM	A direct manipulation interface for cartographic program generation, including LLM programming
Bio	An interactive program synthesis tool for biological analyses
Notebook	A programming system to scope variables in Jupyter notebooks.
TextProc	A word processing system with user-programmable interface components

declarative sentences about how the present system works, we found it helpful to brainstorm the language for our features by thinking about how our current system was different than others.

Step 4: Put the PEA to Work. Now that we have a place to start, the real work can begin. In the sections below, we describe how we have found a PEA useful to identify opportunities for design (Section 5), compare system designs (Section 6), and guide user research (Section 7).

Step N: Iterate. While these list of steps are a helpful way to get started, they should not imply that each is only done once. Above all, a PEA is a tool for programming systems designers to gradually and iteratively make explicit, critique, and refine their understanding of a system and how programmers use it. Many of the most useful insights from a PEA started when disagreements over phrasing a certain **feature** or **behavior** brought both researchers new clarity about the system and its use, leading to cascading revisions of **features**, **behaviors**, and **actions** and their relationships. Like all redescription in qualitative research, redescribing **behaviors** from **actions** is fraught: there is no guarantee that a **behavior** description evokes the right associations or implications, and so maintaining the connection between an **action** and its **behavior** (and refining when necessary) is essential for researchers to continually confirm and revise the adequacy of both descriptions.

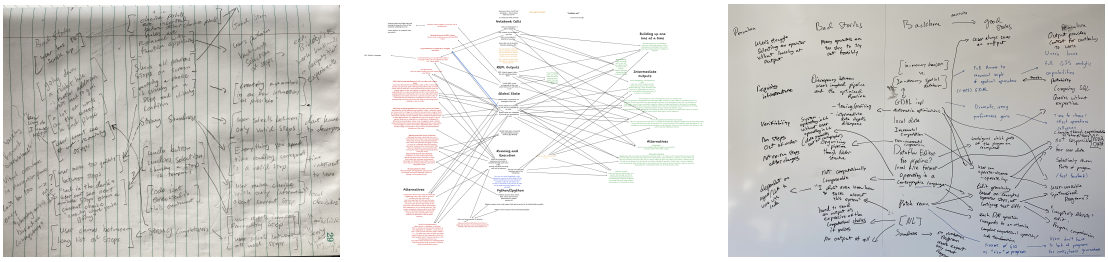


Fig. 3. The diagrams made during a Programming Ecology Activity are complex because the Programming Ecologies they investigate are complex. The diagrams are made over hours of conversation, analysis, and debate. In this paper we present simplified excerpts to illustrate the vignettes below, as the diagrams are not intended to be legible outside of the design team that produced it. These diagrams were from the Bio, Notebook, and MapDM projects, from left to right.

4 Case Studies: What Kinds of Questions Can a Programming Ecology Approach Help Us Answer?

In the sections that follow, we reflect on how Programming Ecology Activities and Diagrams have been useful in collaborations with four groups of researchers across the spectrum of HCI and Programming Languages research.

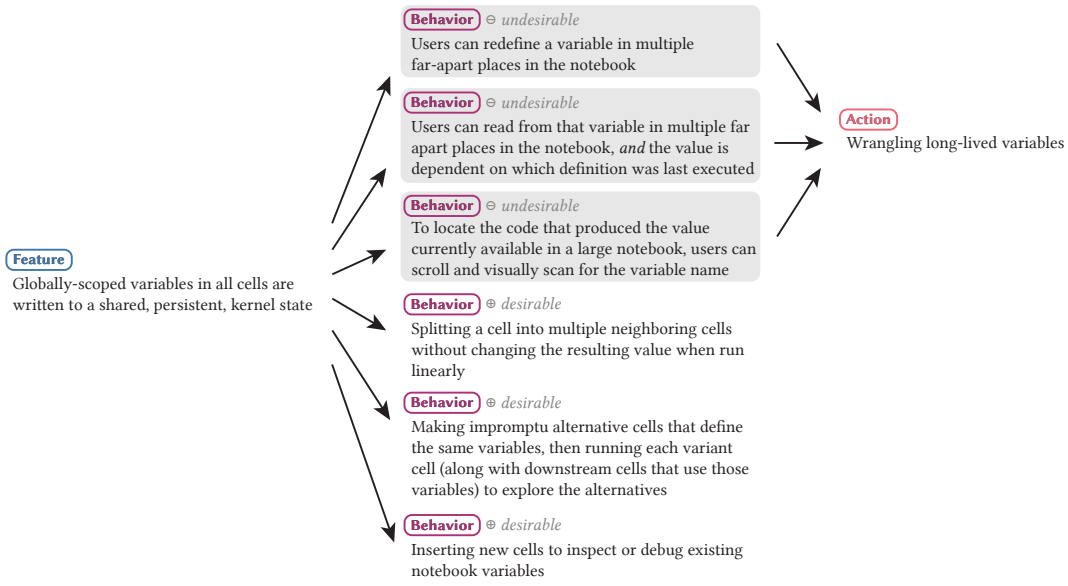


Fig. 4. An excerpt from the Programming Ecology Activity Diagram developed in the Notebook case study, illustrating the vignette in Section 5.1.

The first author facilitated each PEA, first briefly introducing collaborators to the concepts of a PEA and discussing preliminary goals for the session, and then serving as an interlocutor on the research team, participating alongside collaborators in the PEA. As we discuss in Section 8.1, we found that dialogue was a crucial part of the PEA.

5 Identifying Opportunities for Design by Making Connections between System Features and Behavior-Level Descriptions

Once a programming systems designer has findings about challenges that users face with a system, understanding which features are responsible and could be changed without disrupting other user outcomes is often a significant endeavor by itself. In this section, we discuss how a PEA has been useful in our collaborations to frame design questions on specific features, make informed choices about the features that should not change by assessing tradeoffs, and discover how existing features support more desirable behaviors than previously realized.

5.1 Retaining desirable behaviors while eliminating undesirable behaviors

The same **features** in a programming system can be necessary for both desirable and undesirable **behaviors**. A PEA can help designers reframe their understanding of these features to identify design opportunities that make undesirable behaviors difficult or impossible, while retaining desirable behaviors.

In the Notebook project, the researchers wanted to address a common frustration among Jupyter notebook programmers, the many confusions associated with managing variables that remain available throughout a long notebook. The researchers eventually centered their analysis on the **action** "wrangling long-lived variables," an **action** that started quite vague and ambiguous, since no particular **behaviors** had been connected to this **action** so far. To start, the

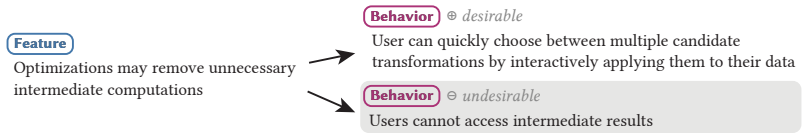


Fig. 5. An excerpt from the Programming Ecology Activity Diagram developed in the MapDM case study, illustrating the vignette in Section 5.2.

researchers used existing user studies of notebook programmers to generate both desirable and undesirable **behaviors** and to generate the **feature** descriptions necessary for describing these **behaviors**.

From the "wrangling long-lived variables" **action**, the researchers generated three PEA undesirable **behaviors** (Figure 4, top): 1. "Users can redefine a variable in multiple, far-apart places in the notebook"; 2. "Users can read from that variable in multiple, far-apart places in the notebook, and the value is dependent on which definition was last executed"; 3. "To locate the code that produced the value currently available in a large notebook, users can scroll and visually scan for the variable name." Researchers observed that these undesirable **behaviors** relied on one prominent **feature** (among others): "globally-scoped variables in all cells are written to a shared, persistent, kernel state.""

However, because the diagram depicted both desirable and undesirable **behaviors**, the researchers learned that many desirable **behaviors** also rely on this same **feature** (Figure 4, bottom): 1. "Splitting a cell into multiple neighboring cells without changing the resulting value when run linearly"; 2. "Making impromptu alternative cells that define the same variables, then running each variant cell (along with downstream cells that use those variables) to explore the alternatives"; 3. "Inserting new cells to inspect or debug existing notebook variables." All of these desirable **behaviors** require that multiple cells can read and write to the same variables. Given the notebook **features** available in standard notebooks, the **feature** that made these desirable **behaviors** possible was "globally-scoped variables in all cells are written to a shared, persistent, kernel state."

To arrive at an opportunity for design, the Notebook researchers asked whether they could make the undesirable **behaviors** impossible while leaving the desirable **behaviors** unaffected. They used a PEA to frame a question: how can the **feature** "globally-scoped variables in all cells are written to a shared, persistent, kernel state" be *broken apart* into two statements, one of which is necessary for the desirable **behaviors**, and the other for the undesirable **behaviors**? Importantly, how can this **feature** be broken apart to identify the smallest intervention in the Python language, IPython kernel, and JupyterLab interface that would have these consequences for **behavior**? This question was the jumping off point for the brainstorming that ultimately produced the Notebook researchers' new programming system. (The system they arrived at made targeted interventions across all three.)

5.2 Weighing The Value of Features That Are Required for Both Desirable and Undesirable Behaviors

When system **features** cannot be broken apart to separate **features** necessary for desirable behaviors from undesirable ones, programming system designers need to assess whether the there is a net benefit to users.

In the MapDM project, researchers developed program optimizations enabling execution of expensive geospatial data transformations at interactive speeds. In our PEA session, we identified together that "system shows the result of a transformation at interactive speed" (**feature**) was required for a **behavior** of "user can quickly choose between multiple candidate transformations by interactively applying them to their data." While this was an important **behavior** for the researchers to support, we realized this new feature also made an existing desirable **behavior** ("user can access

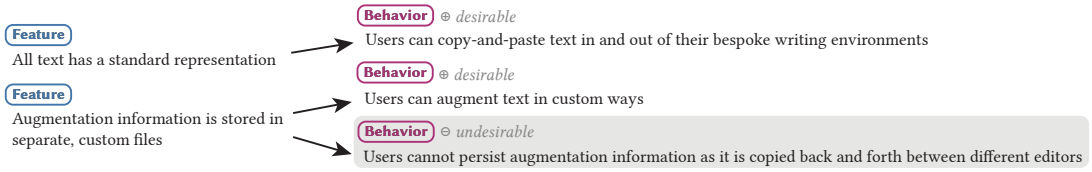


Fig. 6. An excerpt from the Programming Ecology Activity Diagram developed in the TextProc case study, illustrating the vignette in Section 5.2.

intermediate results") *impossible* after refining it to the more specific **feature** of "optimizations may remove unnecessary intermediate computations" (Figure 5). Rather than try to support both behaviors (an ambitious research project on its own), the PEA gave MapDM researchers a place to frame this tradeoff explicitly, and make a choice about the system design while being more cognizant of its implications.

In the TextProc project, researchers wanted users to be able to copy-and-paste text in and out of their bespoke writing environments (**behavior**), and so represented all text in the system in a standard representation (**feature**). Because their system also enabled users to augment text in custom ways (**behavior**) they stored this augmentation information in custom files (**feature**). In a PEA done while the system was being piloted in small observation sessions, we realized together that if text augmentation data is stored in a separate, custom file (**feature**), then users would not be able to persist this kind of text information as it is copied back and forth between different editors (undesirable **behavior**). The researchers found it helpful to highlight the potential **behaviors** their **features** made possible and impossible before the system was fully deployed, both to make a more informed choice and to prepare to change their system if this undesirable **behavior** was found to be more important to users than they anticipated: "at least we know why [this undesirable behavior is a consequence of our system features] when we put this out into the world."

5.3 Making Visible Unexpected Connections Between Features and Behaviors

Often, the links between programming system features and user behaviors can be so manifold and complex that even the creator of the programming systems misses some connections. A PEA can help surface some of the more subtle of these connections. For example, in one PEA session, the lead Bio project researcher made an unexpected connection between a **feature** that ensures users of the system do not create invalid programs and the behavior of selecting between possible next steps while only being shown information about the current, local choice. Users would *not* be able to do this local reasoning if they did not have the guarantee that they could not create invalid programs; because any options the users can select between are guaranteed to be en route to a valid solution, users can make the decision using only local information and not need to reason about the rest of the program to ensure that their solutions are valid. For the Bio project researcher, making this connection gave a precise articulation of why their system enabled "local" reasoning (previously an **action**-level description, but now a **behavior**-level description tied to a set of specific features) and gave them clarity into how their system works as a whole.

6 Comparing System Designs: How do System Variants Change which Behaviors are Possible?

During Programming Ecology Activities, researchers explored system variants by inserting or revising a **feature** description to reflect a hypothetical change to their programming system. Then, for each **behavior** already identified, researchers could assess how the new feature would interact with that **behavior**: For desirable **behaviors**, do they

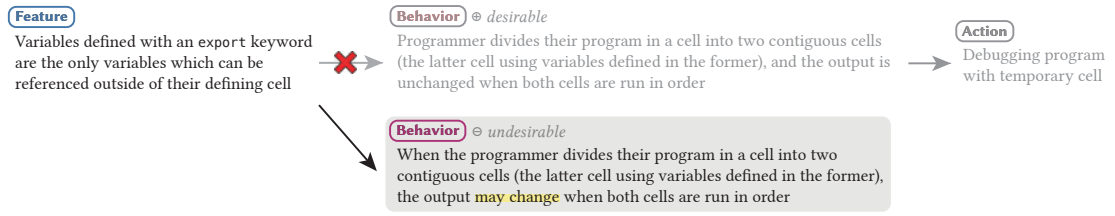


Fig. 7. An excerpt from the Programming Ecology Activity Diagram developed in the Notebook case study, illustrating the vignette in Section 6.1.

remain possible? For undesirable **behaviors**, are they now impossible? While this is no guarantee that an intervention would lead to particular **actions** being observed from future users, screening new system interventions in a PEA enabled researchers to rule out many designs at the ideation stage. Programming Ecology Activities also helped some researchers identify system interventions that could make multiple undesirable **behaviors** impossible which were previously thought to be unrelated.

6.1 Does a Proposed System Design Make Desirable Behaviors Impossible?

In the Notebook project, researchers imagined many design interventions intended to reduce the confusion that Jupyter Notebook programmers had with managing many persistent variables. We will focus on just one here: change Jupyter Notebooks to scope persisted variables to just their individual cell by default, and require an `export` (analogous to `global` or `nonlocal` in Python) keyword to make those variables available to neighboring cells. Framed as a revision to the **feature** descriptions in a PEA (Figure 7), researchers realized that this intervention was not worth building: the PEA made clear that this design would make desirable **behaviors** that the researchers aimed to preserve impossible.

If variables defined with an `export` keyword are the only variables that can be referenced outside of their defining cell (**feature**), then notebook programmers could no longer split a cell in two and have the output be unchanged when both cells are run in order (desirable **behavior**) if a variable defined in the top cell is referenced in the bottom cell. Similarly, notebook programmers could no longer add a temporary cell to print the variables defined in the cell above (**behavior**) in order to debug their programs with a temporary cell (**action**). While we could imagine hypothetical programmers achieving a *different* behavior of splitting cells while marking necessary variables with `export` and adding `export` to any variables they would then print out in a cell below, further user research would be required to understand whether programmers find these new **behaviors** useful and in the same way as the original Behaviors. For the researchers, seeing that this design intervention would interfere with this desirable **behavior** (among others) was enough to remove it from consideration and focus on other designs that did not have these consequences.

From just a description of the potential intervention in terms of a modification to the system **features**, the researchers could rule out this proposal as a solution by showing that it made some valued user **behaviors** impossible. Separating **behavior**-level descriptions from **action**-level descriptions enabled them to clearly assess how a feature change made some **behaviors** impossible to enact, while making clear where further empirical evidence would be required.

We can also extend this analysis to assess other existing systems that address similar problems. Reactive notebooks such as Observable [45] and Marimo [37], for instance, aim to address persistent variable confusion by enforcing that each variable is only defined once (**feature**). While this feature change makes impossible that a user can reference a variable that is ambiguously defined in two places in a notebook (**behavior**), it also makes a desirable behavior

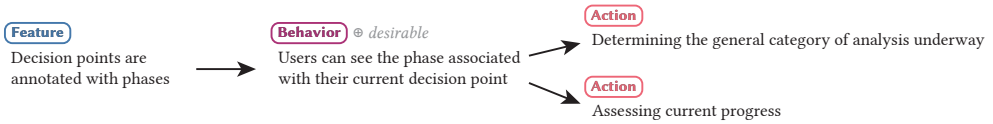


Fig. 8. An excerpt from the Programming Ecology Activity Diagram developed in the Bio case study, illustrating the vignette in Section 6.2.

impossible: researchers found that users valued creating impromptu versions (**action**), from which they generated a behavior description of "defining and selectively executing neighboring cells which define the same variables," which becomes impossible in a reactive notebook. Further user research on these systems would be required to understand the new desirable behaviors this feature change makes possible, at which point designers could assess tradeoffs.

6.2 Identifying How Proposed System Designs Can Solve Unexpected Problems By Making Other Undesirable Behaviors Impossible

When system interventions are proposed as changes to feature descriptions in a PEA, we discovered that the systematic process of re-evaluating each connection in the PEA with the new feature reveals that some proposed features can actually address multiple problems that the researcher had not previously realized were connected. For example, the Bio project researcher faced two challenges in their system:

(1) There may be many different paths to completing a particular analysis, so there is no one overarching workflow or schematic that users can use to situate their current decisions in a larger context. This sometimes led users to feel "disoriented" as they made their decisions and unable to determine the general category of analysis underway.
(2) Because the system does not know what program the user wants to construct, the system does provide a "progress bar" or any other measure of progress to the user. This sometimes rendered users unable to gauge how much time and energy they would need to devote to a particular analysis.

For the first issue, the Bio project researcher considered a few different solutions, including, for example, sampling possible completions of the program at hand and displaying an *average remaining number of decisions* to the user. One of these solutions included annotating numeric *phases* to each decision point and statically ensuring that each step must be monotonically non-increasing in phase, then showing this phase value as an approximate progress bar. Before the PEA, it was not clear which of these proposed solutions would be best; the sampling approach was simpler and required no additional metadata on decision points, but the phase approach was fully deterministic.

However, when integrating the "decision points are annotated with phases" **feature** into the diagram, the Bio project researcher realized that the phase approach could *also* solve the second problem above; if phases were assigned to decision points in the same conceptual register as the biologists are already thinking, then a **behavior** of "users can see the phase associated with their current decision point" could be a redescription of both **actions**: "determining the general category of analysis underway" and "assessing current progress" The PEA helped the Bio project researcher identify a cross-cutting design opportunity that justified the addition of phases to the system, resulting in fewer ad-hoc solutions to design problems.

6.3 Filling Out User Observations for System Interventions

A Programming Ecology Activity is a design tool intended to be in close conversation with user research. When system interventions are built, researchers can use a PEA to clarify how user findings relate to system features.

In the Notebook project, researchers built an intervention to Jupyter Notebooks and observed its use in a user study. One study participant discussed how, prior to the intervention, they organized their notebook code by "putting everything into one big cell." By contrast, the researchers' intervention enabled the participant to incorporate more "process" and "explanation" into their notebook programs. Building from previous PEA sessions, the researchers generated **behaviors** and **features** from these two **action** descriptions. The PEA suggested that because outputs are only displayed at the end of a cell (**feature**), and clustering code into a cell is the primary way code can be visually separated in the notebook (**feature**), this participant was using cells to visually group code (desirable **behavior**) with the consequence that outputs only displayed at the end of each cell (undesirable **behavior**). On the other hand, the researchers' intervention enabled the participant to visually cluster *groups* of cells (**behavior**). Using the PEA, the researchers hypothesized that because their intervention uncoupled the visual organization of code from how outputs were displayed in a notebook, the participant no longer had to choose between visual organization and explanation.

Used as a resource in qualitative research, the PEA helped researchers interpret a user report to generate a theory of how the features of their intervention related to this participant's experience.

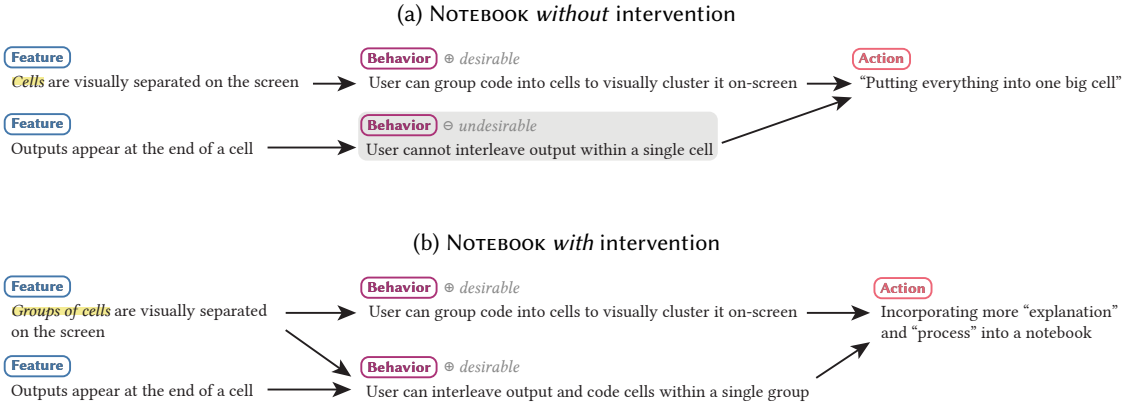


Fig. 9. An excerpt from the Programming Ecology Activity Diagram developed in the Notebook case study, illustrating the vignette in Section 6.3.

7 User Study Design: Using Behavior-level Descriptions to Figure Out What User Study Questions to Ask, What to Look For In Qualitative Analysis, and What to Measure Quantitatively

As Programming Languages and Human-Computer Interaction researchers, we are often in a position to use the systems we build not just as a way to solve a problem, but as a way to ask new questions about the world. But moving from a table-stakes claim like "users of our system are X% faster" to generalizable research questions that are interesting to the community in their own right can be challenging. Ablations, longitudinal studies, and participatory or in-situ approaches are excellent ways of achieving richer forms of knowledge, but what research questions should we ask with these approaches? And once we've decided on a research question and study design, what kinds of quantitative and

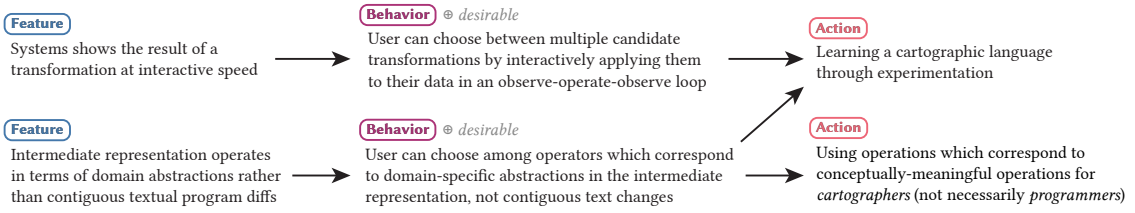


Fig. 10. An excerpt from the Programming Ecology Activity Diagram developed in the MapDM case study, illustrating the vignette in Section 7.1.

qualitative information should we collect? While not a panacea, we have found that Programming Ecology Activities helped researchers operationalize **action**-level user outcomes into **behavior**-level descriptions that can ground a research question, propose theories and corresponding quantitative metrics to collect, and tease apart qualitative relationships between **action**-level user outcomes and programming system **features**.

7.1 Specifying Behaviors from Actions to Frame Research Questions

From an earlier contextual inquiry of cartographers and geospatial analysts, MapDM researchers found that many users learned about geospatial transformations by applying them to their own data and observing the results (*e.g.,* the difference between a Spatial Join, a Merge, and a Union). Researchers wanted to better support this interaction in the MapDM system, so they developed new ways of optimizing these geospatial data transformations to support faster feedback. Then, the researchers wanted to explore how these optimizations changed programming with MapDM in a user study, but felt unsure that an in-lab study would enable them to explore the user outcomes they were interested in, and were trying to develop a more concrete question for a longitudinal study.

In a PEA, we diagrammed the MapDM system with observations from a previous in-lab user study (excerpt in Figure 10). We identified that "system shows the result of a transformation at interactive speed" (**feature**) was required for "user can choose between multiple candidate transformations by interactively applying them to their data in an observe-operate-observe loop" (**behavior**), which was a redescription of "learning a cartographic language through experimentation" (**action**). From this **action**, we also connected another **behavior** that had already been generated: "user can choose among operators which correspond to domain-specific abstractions in the intermediate representation (IR), not contiguous text changes," which was connected to another **action**: "using operations which correspond to conceptually meaningful operations *for cartographers*, not necessarily *programmers*."

When the "observe-operate-observe loop" **behavior** is considered alone, an in-lab study might seem like a viable observation strategy (*e.g.,* "Do users participate in this **behavior** more often?", "Do they succeed at a pre-designed task faster because of this?"). But by making the associated action and its connections explicit, the researchers realized that their question was more complex. If the action of learning through experimentation does not just involve a single "observe-operate-observe" behavior, but is also related to how programmers understand those edits *relating to cartographically meaningful concepts*, a different kind of study might be required. The researchers then used the PEA to frame this research question: "What is the relationship between 1. this operate-observe-operate loop behavior and 2. how the available edits in the interface (given the IR) are cartographically meaningful to them? And how does that relationship impact how programmers use MapDM in learning and reasoning about cartography?"

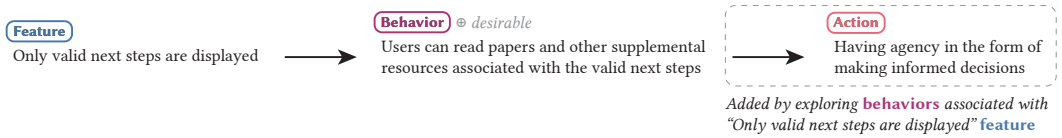


Fig. 11. An excerpt from the Programming Ecology Activity Diagram developed in the Bio case study, illustrating the vignette in Section 7.2.

Phrased in this way, the research question is no longer just about the *computational* understanding that programmers have of the operations in the interface, but the *cartographic* understanding too (and the relationship between them). From their previous user study, researchers had found that participants had struggled to assess the correctness of their interactions (from a spatial analysis perspective) in the interface when they were unfamiliar with their data, suggesting that a longitudinal, in-situ study where programmers are using MapDM on their own data over time would be an appropriate study format.

While the PEA did not prescribe what user study question to ask, researchers used it to "put all the pieces on the table" of their understanding of both the system and the users, and made connections that framed more complicated questions about the relationships between both behaviors and actions that would be more tractable questions for longer-term qualitative user research.

7.2 Operationalizing Questions about the Impact of System Features on User Outcomes with Behavior-Level Descriptions

The Bio project researchers wanted to build a system that would improve experimental biologists' agency in performing computational analysis (an **action**-level user outcome). They had previously hypothesized that the introduction of a certain formal guarantee about the validity of the choices provided to the user would improve user agency ("only valid next steps are displayed" **feature**), and therefore were planning an ablation study of the Bio system with and without this **feature** present. Unlike the MapDM researchers in the previous section, the Bio researchers already knew what **feature** they wanted to ablate in their user study; but, like the MapDM researchers, the Bio researchers only had a fuzzy idea of what it meant to "improve user agency" as a user outcome. Having just performed a PEA, though, and because the Bio researcher knew what system **feature** they wanted to ablate, the Bio researcher could look through the connections they made between **features** and **behaviors** to hypothesize what kinds of effects they might expect to see in the absence of that **feature**, and which of those might operationalize a lack of agency.

One connection of interest was the connection of the **feature** "only valid next steps are displayed" to the **behavior** "users can read papers and other supplemental resources associated with the valid next steps" (Figure 11). If the "only valid next steps are displayed" **feature** were ablated, then users would no longer be sure that the papers and other resources they would be reading would correspond with meaningful biological choices. If that were the case, we might hypothesize that the users would be less engaged with these resources (*i.e.,* only skimming them or not reading them at all) because they might often be a waste of their time to read, and consequently that they would be less able to make informed decisions as they are using the Bio tool. One facet of user agency could then be the users reading, understanding, and choosing between next steps as expressive of meaningful biological choices (and not nonsense), with a qualitative analysis focused on the extent to which users are deeply engaging with the resources associated with each step. The PEA does not immediately read off a comprehensive definition of "agency" nor a concrete user

question, but it did help concretize part of what user agency would look like in this particular context and, consequently, a direction for qualitative observation to supplement simply asking users if they felt agency while using the tool.

7.3 Using Behaviors to Theorize How Quantitative Measures Correspond to Actions

In the Notebook project, researchers built a system that aimed to alleviate global variable confusions of Jupyter Notebook programmers. Alongside an in-situ user study where Notebook programmers used this intervention in their own work, the researchers were curious about what they could learn from logging users' interactions during the study period. They faced an immediate problem: what would be meaningful to measure about Notebook programmers to learn something about global variable confusion?

The researchers realized, however, that they had already generated a detailed theory of what was happening in the programming system when programmers were confused by global variables: the **behaviors** in the PEA. For example, one undesirable **behavior** generated was "Users can define new variables (*e.g.,* `df_import`, `df_cleaned`, `df_analysis`, *etc.*) instead of redefining the same variable to make unambiguous where a persisted variable was last defined." Researchers hypothesized that if their new system was successful, it might remove the need for this **behavior**. Researchers therefore hypothesized that the number of times a variable is defined in a notebook would decrease with the use of their system.

While fully operationalizing "the number of times a variable is defined" leaves its own challenges to be addressed (*e.g.,* how should two notebooks be compared if they have a different length and define a different number of variables?), the PEA gave researchers a foundation to theorize why a change in this quantitative measure should reveal something about their concept of interest.

8 Discussion

8.1 Building Understanding Within A Design Team

The vignettes presented above report on fragments of the iterative process of conversation and revision that is the real work of a PEA. By the conclusion of our case studies it was clear to us and our collaborators that the value of the PEA was in how it occasioned a careful and effortful *working-over* of our understanding, not in the diagram produced, which served primarily as a record and medium for this work. Overall, the purpose of a PEA is less to have a diagram that communicates a system to others and much more to engage in an activity that supports researchers in understanding their own system of choice. As one collaborator said: "I wasn't totally convinced [the PEA] would [work for my project]. Now I'm like, 'Oh, now I get it' having done it. I think it was hard for me to see how it was going to work until I operationalized it in a context I know super well." We labeled the PEA an "activity" because we want to emphasize that it is something researchers engage in, and is not a set of instructions to follow or a diagram that comes out at the end. Like participatory methods more broadly, learning to apply the concepts of Programming Ecology can take more time and practice than simply learning the words [15, p. 21] [39, p. 20], and might be easier with the guidance of a more experienced practitioner. At the conclusion of a PEA, some of our collaborators speculated that they would have had more difficulty learning to do a PEA alone, which is certainly not unique among HCI design techniques.

Although the diagram produced during a PEA is not necessarily a useful artifact for communicating to others outside of the PEA team, we found the activity very useful for communication *within* a PEA team. Many of the most meaningful moments of insight began when a word was used differently between team members, and then trying to find the right words together. To follow Pelle Ehn's use of Wittgenstein in systems design [11, p. 108], we think of the diagrams created in a PEA not as a "model" of a system interaction but as a place to invent and learn a new way of talking about

the system-in-use; in other words, learning to play a new *language-game* [62, §23] together. Overall, we found the process of building the diagram more useful than the output diagram itself, and within a design team, we found PEA supported communication and building a shared vocabulary.

8.2 Designing Far-Reaching Interventions to Programming Ecologies

A recurring idea in discussions with our collaborators was how the PEA encouraged a more holistic picture of programming systems design. Putting different parts of a designer's system-interaction theory in conversation led to surprising connections and new insights. The PEA also seemed to help designers brainstorm a class of system changes that they reported as difficult. In contrast to self-contained interventions (*e.g.*, adding a specific interface button that has minimal interactions with other parts of a system), PEA collaborators reported they could reason about interventions that would have many far-reaching consequences in the system, often at the level of the programming language itself. Collaborators discussed how considering user challenges individually often led them to "band-aids" or "surface-level" changes to a programming system interface—system changes that solved discrete user problems but did not feel integrated into the *ecology* of other features and behaviors.

Addressing user needs with small, atomic additions to a programming system certainly has its place, but our PEA collaborators already felt confident in their ability to design for these kinds of interventions. The designs that challenged them were often focused on interventions that would have sweeping consequences for the rest of the system. Especially as programming systems researchers, they wanted to explore radically new ways to support programming, while remaining responsive to the needs, goals, and work practices of real programmers.

We suspect Programming Ecology Activities will have the most use in designing interventions that will have far-reaching consequences, interacting with many system features and with many different user activities. Thus we see PEA as complementary to other design approaches like User-Centered Design [17], Heuristic Evaluation [44], Cognitive Walkthroughs [55], and we expect designers will use them in concert depending on their needs.

8.3 User Research On Programming Ecologies

As discussed in Section 2.1, researchers working on novel PL tools are increasingly incorporating user research techniques to understand how users interact with programming systems; they incorporate short observation sessions and in-lab user studies to understand the use of programming systems. This has been very exciting to see, but if a key aim of programming systems is to help users express novel computation with a powerful language, other methods might be more appropriate. Investigating the diversity of ways programmers make use of a system in their own work, designing systems through observations of entire practices rather than short sessions to better support broad and unforeseen uses, or collaboratively developing a system with domain experts requires other methodological tools in HCI. In-situ [51] and participatory methods [15, 56, 63] are well-suited for this kind of research, but studying users to inform programming system design and shaping research questions for these kinds of methods is difficult.

We hope that Programming Ecology can help bridge this gap. Although a PEA can help designers across a variety of user study and evaluation methods, we find a particular affinity between PEA and longitudinal, in-situ research. In the Notebook and Bio projects, the PEA helped researchers navigate the rich data these methods provide—taking ideas about programming within the practices of data scientists and biologists and grounding them in behavior descriptions and features. We hope the tools we have presented here support other HCI–PL researchers in studying the role of programming systems not just in the contexts of individual tasks or short observation sessions, but within the larger technical, practical, and social worlds of their users.

In future work we are also excited to explore how a PEA might be used as a site of "mutual learning" [11, p. 332] in a participatory design process in the Bio project. As a bridge between the vocabulary and concepts of programmers (**actions**) and the vocabulary and concepts of programming systems (**features**), the PEA could serve as a place for both researchers and practitioners to articulate ideas across this divide.

8.4 Programming Ecology Activities as Theory-Building

Our case studies revealed how a PEA can help programming systems designers develop a clearer picture of the relationship between a system's features and how users interact it. In other words, the PEA helped designers make a *theory* of their systems more explicit. Programming Ecology is an approach that requires an existing system and user, as the ground of the framework is not a hypothetical system but one sufficiently realized that PEA participants can make determinate claims about **features**, **behaviors**, and **actions**. In this sense, a Programming Ecology Activity is a tool for understanding the theory "embodied" [5] in a system, as Carroll and Kellogg phrase it. Following their suggestion, we might see the programming-system-in-use as the "nexus" from which theories of programming system interaction emerge, and the PEA as a tool for navigating that nexus.

For Peter Naur (building from Ryle [52]), a "theory" [43] refers to an intelligence that "transcends" any documentation about it, because it necessarily includes a practitioner's ability to explain the relationship between the documentation and the activity it describes and adjust it as needed. The diagrams produced through the PEA do not *capture* the theory of the program in documentation, but they serve as artifacts that can be used by researchers in future PEAs to share and develop their theories of a system. We might speculate that as more systems, including programming systems, are programmed with LLM-based tools, which generate code that no human may ever read, we need more powerful tools for maintaining theories of our systems now more than ever.

Carroll and Kellogg considered artifacts as nexus of *psychological* theories (with the predictive and explanatory aims of those theories). Programming Ecology, by contrast, orients designers to reason about the possibility and impossibility of **behaviors** given certain **features**—aiming not to predict or explain programmer behavior, but reason generally about what a programming system makes possible and why, in light of observed user interaction. For systems that aim to support users in expressing novel computational behavior, we have found this to be a productive register for theoretical development.

8.5 Making Better Arguments About Programming Systems Research

In addition to supporting *theories* of programming systems, we also discovered in our case studies how a PEA can support researchers in how they present *arguments* about their systems in publications. One of our collaborators was drafting a research paper for their system at the time of the PEA, and said during the session: "What I'm noticing is that my design principles are all at different levels, one of them is an [**action**]...and then the next one...is halfway in between [a **behavior** and an **action**]—it's not sufficiently specific [to describe the system intervention] but it's also not sufficiently high-level [to describe why users care about it]." They then looked at an **action**-**behavior**-**feature** chain in their diagram to read it off as a single sentence that would also be the structure of a section in their paper: "We wanted to support [**action**] therefore [**behavior**] therefore [**feature**]." Especially when building programming systems, the connection between a system design choice and a generalizable design principle can be distant, and the PEA provided concepts to communicate that connection more clearly in a research paper.

A PEA can also help make more explicit the full value of a research contribution. Another collaborator said: "What I think is really interesting is seeing how even the things that are linked to [new **feature**], for example, are things

we don't really talk about in the [new **feature**] paper... [the PEA] makes clear that that [new **feature**] has a lot of other potential benefits that never made it into the paper. And now I look at it and I think, 'well that's a bummer!'" By reading a PEA diagram as an *argument* about a system, researchers might better communicate the entire *ecological* value of a system intervention.

8.6 What *Isn't* a Programming System? Applying a Programming Ecology Approach Beyond Textual Programming

The case studies described in this paper all featured systems in which users create textual programs—clearly "programming systems." But programming systems can encompass a much broader range of systems.

At the broadest level, we might adopt a functionalist definition of programming system as a system that, in addition to supporting users in "saying common things," also aims to support them in "saying uncommon things too" [6]; in other words, systems that support users in defining highly novel computation unanticipated by the designers.

For example, we might imagine taking a Programming Ecology perspective on Photoshop as a wide range of features that all work together to support users in "saying uncommon things." Consider some hypothetical **behaviors** that are possible with the *layer* **feature** in Photoshop (alongside other **features**, of course): layers can be used filter images, superimpose images, or mask subsequent layers. If we wanted to design a change to layers, we might find a Programming Ecology perspective useful: studying users to get a wide range of **actions**, generating **behaviors**, then look at all the **behaviors** for which layers are necessary. For any change, we could evaluate which **behaviors** become impossible, weigh the tradeoffs of the new design, and hypothesize what kinds of workflows would be disrupted and enabled by the change. By thinking about layers as a single piece in a much larger Programming Ecology, we could start to build the understanding required for changing such a fundamental part of the system without unintended consequences.

While far from a full PEA, this rough sketch hopefully illustrates that seeing a system *as a programming system*, and more specifically *as a programming ecology*, might be useful to researchers building systems that do not include textual programming languages. HCI work in Substrates [35] and Toolkits [16, 31], for example, are focused on supporting users with powerful and diverse interactions based on simple rules and primitives. A Programming Ecology perspective might aid researchers in those areas in studying deployments of their systems, reasoning about design interventions, and scaffolding user studies that investigate how the features of their systems combine to support a variety of interesting interactions.

9 Conclusion: Having Better Conversations About Programming Systems Within A Design Team and Across the Research Community

In this paper we presented a set of concepts for thinking *ecologically* about programming systems and how they are used. Through a design exercise developed across four case studies, the Programming Ecology Activity (PEA), we proposed a way of using these concepts to identify opportunities for design, reason about design interventions, and plan user research.

Programming systems are complex. Often it is only a system's expert users who have sufficient expertise to understand why the system's design is interesting or effective. In each of our collaborations, however, we began a PEA with little or no understanding of the system itself, and yet we found ourselves having generative, concrete, and fascinating conversations that taught both us and our collaborators something new about their system. One collaborator recognized this before we did: "I used to think that meaningful conversation about the design of my system would only be possible with someone who was also in the codebase alongside me, but doing the [PEA] was really productive even though

[the first author] had very little context about the system at the beginning." Beyond building better theories and better arguments about programming systems, our wish for this work is that it helps the HCI–PL community have better programming systems *conversations* as well. And to do that, we have to get into the weeds.

Acknowledgments

Ethics and Privacy Statement

Each of our collaborators consented to their project's inclusion in this paper and approved the accounts of their experiences that are presented. Quotes were included with explicit consent. User studies discussed from collaborating projects were conducted with IRB approval from the associated institutions.

References

[1] Alfred V. Aho, Monica S. Lam, Jeffrey D. Ullman, and Ravi Sethi. 2011. *Compilers: Principles, Techniques, and Tools* (2nd edition ed.). Pearson.

[2] Virginia Braun and Victoria Clarke. 2019. Reflecting on reflexive thematic analysis. *Qualitative Research in Sport, Exercise and Health* 11, 4 (Aug. 2019), 589–597. doi:10.1080/2159676x.2019.1628806

[3] Susanne Bødker and Clemens Nylandsted Klokmose. 2011. The Human–Artifact Model: An Activity Theoretical Approach to Artifact Ecologies. *Human–Computer Interaction* 26, 4 (Dec. 2011), 315–371. doi:10.1080/07370024.2011.626709

[4] Susanne Bødker, View Profile, Clemens Nylandsted Klokmose, and View Profile. 2012. Dynamics in artifact ecologies. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*. 448–457. doi:10.1145/2399016.2399085

[5] J. M. Carroll and W. A. Kellogg. 1989. Artifact as theory-nexus: hermeneutics meets theory-based design. *ACM SIGCHI Bulletin* 20, SI (March 1989), 7–14. doi:10.1145/67450.67452

[6] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (Aug. 2021), 98–106. doi:10.1145/3469279

[7] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376729

[8] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2020. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. http://arxiv.org/abs/1912.04719 arXiv:1912.04719 [cs].

[9] Nils Dahlbäck, Arne Jönsson, and Lars Ahrenberg. 1993. Wizard of Oz Studies: Why and How. In *Proceedings of the 1st International Conference on Intelligent User Interfaces (IUI '93)*. Association for Computing Machinery, New York, NY, USA, 193–200. doi:10.1145/169891.169968

[10] Matthew C. Davis, Emad Aghayi, Thomas D. Latoza, Xiaoyin Wang, Brad A. Myers, and Joshua Sunshine. 2023. What's (Not) Working in Programmer User Studies? *ACM Transactions on Software Engineering and Methodology* 32, 5 (July 2023), 120:1–120:32. doi:10.1145/3587157

[11] Pelle Ehn. 1990. *Work-Oriented Design of Computer Artifacts*. L. Erlbaum Associates Inc., USA.

[12] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78. Springer, Berlin, Heidelberg. doi:10.1007/3-540-09724-4

[13] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (March 2021), 7–14. doi:10.1109/MCSE.2021.3059263 Conference Name: Computing in Science & Engineering.

[14] T. R. G Green. 1989. Cognitive Dimensions of Notations. In *People and Computers V*.

[15] Joan M. Greenbaum and Morten Kyng (Eds.). 1991. *Design at work: cooperative design of computer systems*. L. Erlbaum Associates, Hillsdale, N.J.

[16] Saul Greenberg. 2007. Toolkits and interface creativity. *Multimedia Tools and Applications* 32, 2 (Feb. 2007), 139–159. doi:10.1007/s11042-006-0062-y

[17] Jan Gulliksen, Bengt Göransson, Inger Boivie, Stefan Blomkvist, Jenny Persson, and Åsa Cajander. 2003. Key principles for user-centred systems design. *Behaviour & Information Technology* 22, 6 (Nov. 2003), 397–409. doi:10.1080/01449290310001624329

[18] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering* 19, 5 (Oct. 2014), 1335–1382. doi:10.1007/s10664-013-9289-1

[19] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, New Orleans LA USA, 281–292. doi:10.1145/3332165.3347925

[20] Jon Hindmarsh, Christian Heath, Dirk Vom Lehn, and Jason Cleverly. 2005. Creating Assemblies in Public Environments: Social Interaction, Interactive Exhibits and CSCW. *Computer Supported Cooperative Work (CSCW)* 14, 1 (Feb. 2005), 1–41. doi:10.1007/s10606-004-1814-8

[21] K. Holtzblatt, H. Beyer, and an O'Reilly Media Company Safari. [n. d.]. *Contextual Design*. Elsevier Science & Technology. https://books.google.com/books?id=_BR1cgAACAAJ

[22] Ruanqianqian Lisa Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1243–1255. doi:10.1109/ICSE55347.2025.00232 ISSN: 1558-1225.

[23] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 13. doi:10.22152/programming-journal.org/2023/7/13 arXiv:2302.10003 [cs].

[24] Heekyoung Jung, Erik Stolterman, Will Ryan, Tonya Thompson, and Marty Siegel. [n. d.]. Toward a Framework for Ecologies of Artifacts: How Are Digital Artifacts Interconnected within a Personal Life? ([n. d.]).

[25] J. F. Kelley. 1984. An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. *ACM Transactions on Information Systems (TOIS)* 2, 1 (Jan. 1984), 26–41. doi:10.1145/357417.357420

[26] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–11. doi:10.1145/3173574.3173748

[27] Rob Kling and Walt Scacchi. 1982. The Web of Computing: Computer Technology as Social Organization. In *Advances in Computers*, Marshall C. Yovits (Ed.). Vol. 21. Elsevier, 1–90. doi:10.1016/S0065-2458(08)60567-7

[28] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering* 20, 1 (Feb. 2015), 110–141. doi:10.1007/s10664-013-9279-3

[29] Henrik Korsgaard, Peter Lyle, Joanna Saad-Sulonen, Clemens Nylandsted Klokmose, Midas Nouwens, and Susanne Bødker. 2022. Collectives and Their Artifact Ecologies. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (Nov. 2022), 1–26. doi:10.1145/3555533

[30] Klaus Krippendorff. 1989. On the Essential Contexts of Artifacts or on the Proposition That "Design Is Making Sense (Of Things)". *Design Issues* 5, 2 (1989), 9. doi:10.2307/1511512

[31] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–17. doi:10.1145/3173574.3173610

[32] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming (MSFP 2014)*, Vol. 153. 100–126.

[33] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. doi:10.1145/3408991

[34] Peter Lyle, Henrik Korsgaard, and Susanne Bødker. 2020. What's in an Ecology? A Review of Artifact, Communicative, Device and Information Ecologies. In *Proceedings of the 11th Nordic Conference on Human-Computer Interaction: Shaping Experiences, Shaping Society*. ACM, Tallinn Estonia, 1–14. doi:10.1145/3419249.3420185

[35] Wendy E. Mackay and Michel Beaudouin-Lafon. 2025. Interaction Substrates: Combining Power and Simplicity in Interactive Systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–16. doi:10.1145/3706598.3714006

[36] Scott D. Mainwaring, Ken Anderson, and Michele F. Chang. 2005. Living for the Global City: Mobile Kits, Urban Interfaces, and Ubicomp. In *UbiComp 2005: Ubiquitous Computing*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Michael Beigl, Stephen Intille, Jun Rekimoto, and Hideyuki Tokuda (Eds.). Vol. 3660. Springer Berlin Heidelberg, Berlin, Heidelberg, 269–286. doi:10.1007/11551201_16 Series Title: Lecture Notes in Computer Science.

[37] Marimo, Inc. [n. d.]. Marimo. https://marimo.io/

[38] Microsoft. [n. d.]. Visual Studio Code. https://code.visualstudio.com/

[39] Kathleen Musante (DeWalt) and Billie R. DeWalt. 2010. *Participant Observation: A Guide for Fieldworkers*. Bloomsbury Publishing PLC.

[40] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. doi:10.1109/MC.2016.200

[41] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. doi:10.1145/1015864.1015888

[42] Bonnie A. Nardi and Vicki O'Day. 1999. *Information Ecologies: Using Technology with Heart*. The MIT Press. doi:10.7551/mitpress/3767.001.0001

[43] Peter Naur. 1985. Programming as theory building. *Microprocessing and Microprogramming* 15, 5 (May 1985), 253–261. doi:10.1016/0165-6074(85)90032-8

[44] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems Empowering people - CHI '90*. ACM Press, Seattle, Washington, United States, 249–256. doi:10.1145/97243.97281

[45] Observable, Inc. [n. d.]. Observable. https://observablehq.com/

[46] JOHN F. Pane, CHOTIRAT "ANN" Ratanamahatana, and BRAD A. Myers. 2001. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies* 54, 2 (Feb. 2001), 237–264. doi:10.1006/ijhc.2000.0410

[47] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

[48] Python Software Foundation. [n. d.]. Python. https://www.python.org/

[49] R Core Team. 2023. R: A Language and Environment for Statistical Computing.

[50] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. doi:10.1145/1592761.1592779

[51] Yvonne Rogers and Paul Marshall. 2017. *Research in the Wild.* Springer International Publishing, Cham. doi:10.1007/978-3-031-02220-3

[52] Gilbert Ryle. 2009. *The concept of mind.* Routledge, London ; New York. OCLC: ocn297405035.

[53] Donald A. Schön. 2017. *The Reflective Practitioner: How Professionals Think in Action.* Routledge, United States.

[54] Mary Shaw. 2020. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 1–44. doi:10.1145/3480947

[55] Rick Spencer. 2000. The streamlined cognitive walkthrough method, working around social constraints encountered in a software development company. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems.* ACM, The Hague The Netherlands, 353–359. doi:10.1145/332040.332456

[56] C. Spinuzzi. 2005. The Methodology of Participatory Design. *Technical Communication* (May 2005). https://www.semanticscholar.org/paper/The-Methodology-of-Participatory-Design-Spinuzzi/3cbf8fd37d5502670f61ca06449f14df7a049a51

[57] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4 (Nov. 2013), 1–40. doi:10.1145/2534973

[58] Lucy A. Suchman. 1987. *Plans and Situated Actions: The Problem of Human-Machine Communication.* Cambridge University Press. Google-Books-ID: AJ_eBJtHxmsC.

[59] The Eclipse Foundation. [n. d.]. Eclipse. https://www.eclipse.org/downloads/

[60] Mads Tofte. 1990. Type Inference for Polymorphic References. *Information and Computation* 89, 1 (Nov. 1990), 1–34. doi:10.1016/0890-5401(90)90018-D

[61] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* Association for Computing Machinery, New York, NY, USA, 760–771. doi:10.1145/2884781.2884849

[62] Ludwig Wittgenstein, G. E. M. Anscombe, P. M. S. Hacker, Joachim Schulte, and Ludwig Wittgenstein. 2010. *Philosophische Untersuchungen: = Philosophical investigations* (rev. 4. ed., 3. [dr.] ed.). Wiley-Blackwell, Chichester, West Sussex, U.K Malden, MA.

[63] Dennis Wixon and Judith Ramey. 1996. *Field Methods Casebook for Software Design.* Wiley. Google-Books-ID: PKRQAAAAMAAJ.

[64] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation* 8, 4 (Dec. 1995), 343–355. doi:10.1007/BF01018828

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009