How Statically-Typed Functional Programmers Author Code

Justin Lubin
justinlubin@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

ABSTRACT

How working statically-typed functional programmers author code is largely understudied. And yet, a better understanding of developer practices could pave the way for the design of more useful and usable tooling, more ergonomic languages, and more effective on-ramps into programming communities. The goal of this work is to address this knowledge gap: to better understand the high-level authoring patterns that statically-typed functional programmers engage in. I did a grounded theory analysis of thirteen programming sessions of practicing functional programmers, eight of which also included a semi-structured interview. The theory I developed gives insight into how the specific affordances of statically-typed functional programming affect domain modeling, type construction, focusing techniques, exploratory strategies, mental models, and expressions of intent. The success of this approach in revealing program authorship patterns suggests that the same methodology could be used to study other understudied programmer audiences.

CCS CONCEPTS

• Human-centered computing → HCI theory, concepts and models; **• Software and its engineering → Functional languages**.

KEYWORDS

Static types, functional programming, grounded theory, need-finding, interviews

ACM Reference Format:
Justin Lubin. 2021. How Statically-Typed Functional Programmers Author Code. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '21 Extended Abstracts), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411763.3451515

1 INTRODUCTION

Statically-typed functional programming languages like Haskell, OCaml, SML, Elm, and F# offer features and norms—like expressive type systems, strong static guarantees, and an emphasis on small and reusable functions free of side effects—that differentiates them from other classes of languages. These attributes are different enough from those found in more mainstream languages that they engender distinct modes of interaction between statically-typed functional programmers and their language, environment, and tooling.

The aim of this work is to understand how the specific affordances of statically-typed functional programming affect the way programmers author code. The end goal is to deepen our understanding of an understudied programmer population, lay the foundation for evidence-based design of useful and usable languages and tools, and elucidate tacit community knowledge, which could itself ease the onboarding of new members to the community.

I employed a grounded theory approach [6] to analyzing thirteen statically-typed functional programmers engaged in coding sessions, eight of which occurred live over Zoom and included a semi-structured interview afterward. The remainder of these sessions were drawn from livestreaming websites for increased ecological validity. My theory unifies my observations in a framework consisting of four primary categories: (1) type construction, (2) focusing techniques, (3) hierarchical and opportunistic programming, and (4) mental models and expressing intent.

The methodology I used to develop this theory is generalizable to other understudied classes of programmers. Moreover, the insights generated by this theory can be used not only as a foundation for improved languages and tooling, but also as a starting point for further interdisciplinary investigation into *why* statically-typed functional programmers author code the way that they do, whether or not their strategies could be improved, and whether the behavior of working functional programmers should shape how we teach beginners.

In summary, I observed statically-typed functional programmers write code and interviewed them about their code authoring processes, resulting in:

- A grounded theory of how statically-typed functional programmers author code, including their domain modeling, type construction, focusing techniques, exploratory strategies, mental models, and expressions of intent.
- A template for future need-finding studies of understudied programmer populations.

2 RELATED WORK

Design Patterns. Design patterns are reusable high-level program design templates (e.g., the abstract notion of an "iterator" for sequentially accessing elements), popularized by the influential *Design Patterns* program design compendium [4]. Although they have have seen success in industry for promoting clear communication and reusable code [2], their declarative nature gives little insight into *how* these patterns are achieved or written in the first place.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CHI '21 Extended Abstracts, May 8–13, 2021, Yokohama, Japan
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8095-9/21/05.
https://doi.org/10.1145/3411763.3451515

(a) **Recruited Participants**

ID	Language	Exp. (Years)	Exp. (Kind)
P1	SML	3	Academia
P2	OCaml	9	Academia, industry
P3	OCaml	3	Academia, industry, open source
P4	OCaml	0.5	Academia
P5	Elm	3	Industry, open source
P6	Elm	1	Industry
P7	Reason	2.5	Academia
P8	Elm	7	Industry

(b) **Livestreamers**

ID	Language	Task
L1	Haskell	HTTP requests to GitHub API
L2	Elm	Board game with GUI
L3	Haskell	REST API for cryptography
L4	OCaml	Web server
L5	Haskell	Advent of Code (coding challenge)

Table 1: Recruited Participants and Livestreamers. (Exp. denotes experience with statically-typed functional programming.)

Program Transformations. Low-level code edit actions appear pervasively in code editors today as *automated program transformations* (e.g., tools such as Introduce Parameter and Reorder Definition). Thompson details many such transformations applicable specifically to statically-typed functional programming languages [9], focusing specifically on semantics-preserving refactors. While these transformations provide a detailed characterization of the low-level edits programmers undertake, they do not capture the high-level intent of programmers as they construct code.

Psychology of Program Construction. Between the high-level, declarative nature of design patterns and the low-level descriptions of program transformations lies the gray area of how programmers connect these two extremes: the psychology of program construction. One important distinction in the psychology of programming is between *hierarchical* and *opportunistic* patterns of program construction. Hierarchical programming is characterized by a systematic approach of refining the abstract to the concrete via a plan constructed ahead of time. Opportunistic programming is characterized by deviations from this pattern (caused by, for example, a lack of conceptual clarity or tasks that exceed the capabilities of working memory) [5]. In this study, I am concerned not just with programmers' goals, but the processes by which they achieve these goals. Accordingly, I explore the interplay between hierarchical and opportunistic modalities in the context of statically-typed functional programming, the affordances of which (e.g. expressive type systems and exacting compiler guarantees) have substantial impact on patterns of program construction.

Natural Programming. Myers et al. put forth a research agenda of *natural programming*, or "aiming for the language and the environment to work the way nonprogrammers expect" [8]. My study shares the goal of understanding what is natural to those who write code, but differs in its treatment of pre-existing expertise. What is natural to *expert* statically-typed functional programmers is inextricable from their training and experiences; I am concerned here with the existing authoring patterns that these programmers follow, regardless of their naturalness to those without training.

3 METHOD

Participants and Recruitment. I conducted study sessions with eight participants with a variety of experience backgrounds, focusing mostly on those with at least two years of statically-typed functional programming experience (Table 1a). I screened participants (recruited from personal contacts and Slack workspaces) via a survey for (1) participants with sufficient prior experience with statically-typed functional programming and (2) a diverse set of languages.

Study Protocol. Study sessions consisted of two recorded back-to-back sections of approximately 45 minutes each on Zoom. In the first half, participants narrated their thought processes aloud as they completed tasks in a statically-typed functional programming language of their choice. I selected these tasks (such as implementing a calculator, designing an API for 2D geometry, and performing string manipulations) to be relatively open-ended and adapted them as time went on as a form of theoretical sampling [6] to investigate particular themes. Moreover, as participants completed tasks, I incrementally asked them to complete follow-up tasks to investigate how they would adapt their code. In the second half, I asked participants in a semi-structured interview to talk about their experience with statically-typed functional programming and to elaborate on particular topics that came up during their narration.

Livestreamers. To increase ecological validity and to introduce an element of random diversity into my theoretical sampling, I also analyzed 30–60 minute video clips from five statically-typed functional programmers who publicly livestreamed coding sessions (Table 1b). I took these video clips from Twitch and YouTube.[1]

Analysis. I used a grounded theory approach [6] to analyze the session and livestream recordings. I started by tagging "chunks" of each video with specific summaries. I then grouped these chunks by similarity into open codes and kept track of salient observations by writing them down in memos. Finally, as I continued the data collection, I iteratively built up larger categories until identifying the key categories at play in the programmers' processes.

4 RESULTS

My grounded theory analysis revealed that:

- (§4.1) Participants often started a task by iteratively constructing types to model their problem domain and encode design decisions.
- (§4.2) Participants leveraged types to help themselves focus on the task at hand by relying on the compiler as an assistant and using these types to help them plan and decompose their tasks.
- (§4.3) Sometimes, when faced with a difficult or unknown problem domain, participants complemented this systematic

[1] Twitch: https://www.twitch.tv/. YouTube: https://www.youtube.com/.

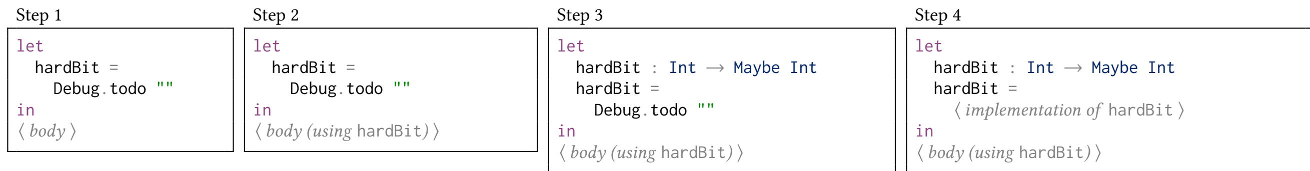

Figure 1: How P8 constructs a "hard bit" of a function. He (1) introduces the hard bit in a let binding, then (2) uses the new variable in the body. This usage helps him (3) deduce an explicit type for the variable, which further helps him (4) implement its definition.

style of programming with a more exploratory one, the details of which were highly varied in comparison to the systematic style.

(§4.4) Lastly, no matter the style of programming, participants had diverse mental models and expressed their intent in many different ways, not all of which produced valid code.

Some of these observations are not necessarily specific to statically-typed functional programmers, but the following sections elaborate on how they play out specifically in the statically-typed functional programming process.

4.1 Type Construction

Using types for domain modeling is an important part of the workflow of statically-typed functional programmers. As P2 said, "That's usually my first instinct ... let me understand my domain, let's write down what these things are, let's give them a name, let's organize them appropriately, *and then* we can start to define our behavior on top of these things." P6 remarked that types help him detect and avoid design decisions *before* writing any code, also adding: "Once you get the types right, the rest of it follows pretty quickly." Given the perceived benefits of this type-driven development style, one might wonder: *How are these types constructed "right" in the first place?* Based on the value that these programmers place on rigorous statically-enforced types, one might get the impression that they are the result of a purely rational process from which the rest of the program flows effortlessly. But the reality is quite a bit messier.

4.1.1 Feeling Odd When Types Are Amiss. At times, participants reported feeling "very wrong" (P4) or that something felt "weird" (P5) or "fishy" (P7) while coding. Ultimately, each time a participant reporting feeling this way, it could be traced to types they viewed as unsatisfactory: in all cases, participants either identified particular discomforting aspects of how they modeled their types, or, after modifying their types, reported being more at ease. One prominent trigger for these feelings was *data representation redundancy*, which both P7 and P8 explicitly called out as particularly undesirable, recalling prior examples of this phenomenon that lead to bugs. As a consequence of this feeling of personal discomfort, participants were able to diagnose and repair issues with their types as they constructed them.

4.1.2 Iteratively Constructing Types and Expressions. For many participants, type construction was not just a linear (or even branching) process, but, rather, a cyclical one, often including the construction of expressions before the types had solidified. For example, participants wrote example expressions *before and during* the construction of their types (P4, P6, P8), basing their decisions of how to structure their types on how they structured their expressions. More iteratively, P7 took a developer-centric viewpoint, noting that although he first thinks "From a simple/conceptual/ontological kind of level: How should these types be?" he sometimes realizes "Oh, okay, wow this is actually way more painful. Like, it felt right to me from just like a lofty, abstract perspective, but now that I'm getting into the nitty-gritty ... ergonomically, it's too costly." Figure 1 depicts the most extreme level of iteration: P8's handling of a "hard bit" of a function, which he pithily characterizes as "[Having] the usage drive the type rather than the type drive the usage."

4.2 Focusing Techniques

> "No, they're pairs! What am I doing? Yeah, ugh, okay—I'm trying to do too many things at once. That's also a pretty common thing." (P3)

Part of the challenge of programming—like any complex task—is managing one's *mental workload*, or "the amount of mental work or effort necessary for a person ... to complete a task over a given period of time," which "is not merely a property of the task, but also of the individual, and their interaction" [10]. Some of the variance between individuals' mental workload for a given task is their ability to manage their *working memory*, a "temporary storage system under attentional control that underpins our capacity for complex thought" [1, Chapter 1]. Accordingly, to complete complex tasks, programmers must focus their attention on relevant information, reducing extraneous mental burden.

Although a more rigorous methodology of mental workload analysis would be required to determine whether the following strategies are actually effective, I observed and participants explicitly noted a few techniques they used to help themselves focus that were made possible by the nature of statically-typed functional programming languages.

4.2.1 Relying on Compiler as Assistant. For many participants, error messages from their compiler served as an auto-updating to-do list. The programmer has a transformation in mind and expresses their intent to complete it by modifying a part of the program. From there, the compiler presents the programmer with a list of sub-tasks to complete the transformation—the programmer can just *follow the errors*. For example, P1 described the process of modifying a type:

> It broke a lot of stuff, but it was super nice because I made the change and the type checker told me: "This no longer

```
-- Set (Position, Position)
-- Set (Position, Tile, Maybe Color, Position)
-- Board, Set (Maybe Color, Position)
-- Board
```

Figure 2: L2 "followed the types" by writing down the types of each step of moving a board game piece in comments (shown here), then, afterward, implementing each one sequentially.

> type checks; here's all the places where it goes wrong." And I can just go through and fix all the type errors, and that essentially is doing the change.

L4 demonstrated the extent to which compiler errors are integral to the development process, noting, in the absence of negative feedback from the compiler, "I'm a little troubled because … I expect the compiler to get mad at me right now, and it's not getting mad at me, and that makes me nervous." P1 echoed this sentiment, noting his discomfort at situations in which the compiler cannot give negative feedback: "Lifting an auxiliary function like that can be fraught, because sometimes the auxiliary function and the original function have the same type … and this is not necessarily something the type checker can fix for you." Participants felt similarly about the use of wildcards in pattern matching, which silently assign preexisting behavior to new variants of an enumeration. P7 mentioned that, in his main codebase, wildcards are *completely disallowed* for this reason, even though they make programming more convenient.

An interesting source of variation between participants was *in what order* they completed the sub-tasks set forth by "following the errors." Mostly, participants completed semantics-preserving refactors to get their code into a clean, working state before introducing new or different semantics—but that was not always the case. For example, when adding a possibly-failing division operation to his calculator, P8 simultaneously made his function return a `Result`[2] *and* implemented new functionality before refactoring other parts of the function to handle the new possible errors. He noted that he did so because he knew exactly how the result handling would work for the division case, but not for the other cases. Similarly, P7 added new semantics to a function *first* as a lightweight way to make sure his type modifications would work out, thereby avoiding committing to an arduous sequence of refactors too early.

4.2.2 Type-Directed Top-Down Decomposition. Participants frequently used types to structure their top-down decomposition of problems, a strategy many participants referred to as *following the types*. L2 exemplified this pattern by writing down the types of each step of a desired algorithm, then later filling in their implementations (Figure 2). P8 took a related approach when implementing a string manipulation function: he knew he needed to use the higher-order library function `List.indexedMap`, but was unsure how to implement the function to pass into it, so he copied and pasted the type signature from the documentation into his code and implemented the function top-down from the signature. P7 stressed the importance of this general strategy in maintaining his focus:

> I find types really help me just organize … it's very important for me to be able to just chunk different parts of functionality, and … once I've gotten down the type signature, I can just focus on this function and not think about really anything outside of it.

Similarly, P3 highlighted that constructing expressions top-down from the type lets him think more clearly about sub-tasks: "Just like, *actually typing this out*, and sitting here, kinda freed my brain of this outer context."

4.3 Hierarchical and Opportunistic Programming

Participants primarily engaged in hierarchical programming by "following the types" (Section 4.2.2); the opportunistic deviations from this pattern, however, were far more varied. These opportunistic strategies often complemented the hierarchical ones harmoniously, with participants frequently and fluidly switching between the two. Guindon observed this harmony in the context of software design [5]; I here describe it in that of statically-typed functional programming.

4.3.1 Opportunistic Strategies. When programming opportunistically, participants engaged in strategies such as:

Using the top level of their file as a blank canvas to sketch code. L1 directly coded a draft of a function at the top level of the file then used it as a reference for a more robust function later. P4 took a similar approach, implementing his calculator function first as a pattern match at the top level and later wrapping it in a function. L4 took a more lightweight approach, using a sketch at the top level only to query complicated type information from the compiler.

Constructing expressions bottom-up. Participants sometimes constructed expressions by iteratively wrapping expressions with more complex structures. In the example above, L1 first implemented the function's core logic in a `do` block, then wrapped the entire expression in a function called `loop` when it called for recursion (as did P4 for his calculator). Similarly, after implementing a function's core logic, P5 wrapped it in a `case` expression that handled errors.

Another common way to build expressions bottom-up was to iteratively construct a "pipeline" (i.e., composition) of functions, especially when the types were concrete. For example, L1 built up a string processing pipeline one function at a time, testing each iteration in the REPL. (P7, P8, and L5 did similarly but without a REPL.)

Performing pattern-matching to handle sub-cases independently. When dealing with abstract types with many existing combinators, participants often introduced pattern matches that would be unnecessary with the use of more idiomatic combinators, but that allowed them to visually examine and handle sub-cases independently. P8 noted an explicit `case` expression is helpful in these cases because "It's telling me, here's all the possibilities, just handle them."

4.3.2 Interplay of Hierarchical and Opportunistic Programming. The above opportunistic strategies were often used in concert with hierarchical strategies. P3 uses both these types of strategies, writing code that meets in the middle:

[2] A `Result` indicates a possible failure that must be explicitly handled; it is a type-safe way of making a function partial.

> I kind of understand, maybe, what I've got, so I can do some bottom-up exploration. And I pretty much know where I want to be (which is the type signatures), and it allows me to do some top-down programming. And when it's not clear to me how to connect the two, and … I'm not feeling super productive or I feel stuck trying to think from one end, I just switch to the other to try to glean some more context.

Although many participants stressed the value of "following the types," sometimes the rigid workflow it entails made it hard for them to overcome indecision. For example, P7 remarked:

> Maybe I already spent a lot of time trying to think about how to do it the right way just in my head and I'm not getting anywhere. Then, at that point, I'm just going to be like, okay, it's time to push through and suppress my perfectionist tendencies of stripping away all redundant things.

When "just pushing through", participants often equivocated between multiple expressions with similar (or identical) semantics, trying to settle on the most ergonomic one for the task at hand. This mentality also manifests in the pattern of writing code not as a final goal, but as a process of clarification: P3 highlighted the importance of planning in his coding process, but also includes pushing through with "exploratory" code in the planning phase. P8 commonly exhibits this behavior by starting with many pattern matches (as in Section 4.3.1) then transitioning the code to a more streamlined form in a bottom-up fashion once it is functional and its patterns are more easily recognizable. He noted that once the `case` expressions are written down, "You can start at the deepest level and sort of streamline them."

4.4 Mental Models and Expressing Intent

Participants had highly diverse mental models, and, although participants relied heavily on their compilers (Section 4.2.1), they did not always conceptualize their code the same way that their compilers processed it.

4.4.1 Diversity of Mental Models. Mental models differed significantly, even within the same programmer. For example, P8 introduced local, argument-passing state to his calculator to implement history, but mentioned that he would handle the task completely differently in a model-view-controller architecture (which he almost always uses while programming in Elm). For the same task, P5 did not even consider a mutable reference, whereas P1 wanted to use one immediately (but ultimately settled on explicit state passing). P4 initially conceptualized the history as a specialized binding context, but decided to use a mutable reference for ease of implementation (which he felt to be unsatisfactory).

This diversity hints at the trade-off between what feels (1) natural, (2) easy, and (3) right. Mutable references were easy yet felt wrong to P1 and P4 in the above paragraph, but were natural to P1 and not to P4. As another example, P3 was working in a context that required an integer, but he realized that the computation might fail; his initial reaction was simply to use −1 to signal failure, but he could not bring himself to do so. (It felt natural and easy, but not right.)

4.4.2 Reasoning about Code Essence. Compilers treat programs as formal objects, but programmers participants often engaged in authoring patterns that demonstrated reasoning about higher-level meaning. Frequently, these patterns did not result in valid code; nonetheless, they were a helpful for participants to concretize and refine their intent.

Conflating a value with its image under a function. Multiple participants identified values in their problem domain with different representations of the same concept, leading to formally invalid but not implausible code. For example, when checking for a division by zero in their calculator, P3, P5, and P8 first checked if the right-hand side of the division was equal to the integer literal `0`, even though it was actually an expression in their grammar.

Checking only for patterns of interest. Participants often used pattern matching to handle one specific case, using a wildcard to discard all others. (Including each case handled by the wildcard would increase the size of the constructed expression dramatically, especially when matching on tuples or using nested patterns.) For example, when working with pairs of computations that might fail, both P1 and P5 immediately introduced a pattern-matching expression and handled the "success" case first, only adding the error handling afterward (using a wildcard as a catch-all).

Although wildcards are helpful in that they reduce the attention needed for error branches, this feature is in direct tension with participants' desires for the compiler to act as an assistant; as mentioned in Section 4.2.1, wildcards prevent the compiler from statically checking that new variants are handled correctly. P1 summarized this issue:

> I have some code where I have a bunch of cases, and then I think I'm at the end … and then I wildcard the last pattern … then I add a new variant to the type, and then suddenly I compile with no errors, but then whenever I use that new constructor, it falls off the end of the function.

Reasoning purely despite effects. Both L1 and L5 used `map` to apply a function to a list, but subsequently realized that the iteration needed to stop early, rewriting the code to handle the "effect" of early termination. L1 reasoned intuitively about an "infinite composition" of a function of type `Maybe ETag → IO (ETag, [Pull])`, then had to figure out what this "composition" looked like with effects.

4.4.3 Signaling and Executing Intent. Participants often had high-level patterns in mind that they needed to translate into executable code. In doing so, there were many details that they needed to work out: some low-level and systematic, and others high-level and involved. For example, reflecting on his process of adding history to his calculator, P1 noted:

> I had a pretty good idea of what I was going to do. The main question was just: How do I represent the state? It seemed pretty clear that the pattern was going to be: thread some state around … then essentially … add a case to the evaluator where, when you hit this case, you dereference the cell and grab the data.

For this task, P2 noted that the type system made him "think, at each layer, who needs to have access to the state and who needs to update it." These high-level plans entail many low-level edit actions (e.g., introducing parameters and updating call sites), but participants initially expressed their intent here via modifications to type signatures.

Sometimes, participants noted that the type system was not expressive enough to capture their intent, mostly when trying to "make illegal states unrepresentable." In such cases, participants signaled their intent that certain states should be illegal by pattern-matching and asserting `False` (P3), throwing an exception (P4), or returning a `Result` type (P5).

5 LIMITATIONS

Two immediate limitations to this study are (1) its small sample size and (2) the fact that it is purely qualitative. Future work should extend the theory with input from more participants and should test the predictive power of the theory. More fundamentally, asking participants to design *and* think aloud can overload their working memory [3]; these sessions could thus be skewed toward opportunistic behavior (although such working memory overload may also be common in a working environment). Another limitation of this study particularly relevant for tool designers is that it captures how statically-typed functional programmers author code in their *current* environments and languages—not necessarily what they would do "naturally." A natural-programming elicitation [7] study with expert statically-typed functional programmers would serve as an interesting comparison of an "ideal" world to the world as it is now.

6 CONCLUSION

This study deepens our understanding of how statically-typed functional programmers author code. Future work could build on this knowledge to design evidence-based tools that leverage existing program authoring patterns. For example, a tool cognizant of how statically-typed functional programmers specify intent (Section 4.4.3) could leverage familiar task decomposition strategies (Section 4.2.2) to serve as a better programmer's assistant than standard compiler error messages. Or a tool could support the expressiveness of wildcard patterns (Section 4.4.2) while instilling the confidence of compiler feedback (Section 4.2.1) by informing programmers of matching wildcards when they introduce new enumeration variants and using program synthesis to provide suggestions. These insights suggest that this same methodology could be applied to other programming audiences where we currently lack need-finding analyses, ultimately leading to more accessible and expressive tooling for understudied communities.

ACKNOWLEDGMENTS

I thank Sarah Chasins for her excellent mentorship and guidance on this project, and I thank Ravi Chugh for the insightful conversations that led to the development of the questions in this study. I am also indebted to the anonymous participants of my study for making this research possible. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1752814. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Alan Baddeley. 2007. *Working Memory, Thought, and Action.* https://doi.org/10.1093/acprof:oso/9780198528012.001.0001
[2] Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O. Coplien, Lutz Dominick, and Frances Paulisch. 1996. Industrial Experience with Design Patterns. In *International Conference on Software Engineering (ICSE).* https://doi.org/10.1109/ICSE.1996.493406
[3] Simon P. Davies and Adrian M. Castell. 1994. From Individuals to Groups Through Artifacts: The Changing Semantics of Design in Software Development. In *User-Centred Requirements for Software Engineering Environments (NATO ASI)*, David J. Gilmore, Russel L. Winder, and Françoise Détienne (Eds.). https://doi.org/10.1007/978-3-662-03035-6_2
[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.*
[5] Raymonde Guindon. 1990. Designing the Design Process: Exploiting Opportunistic Thoughts. *Human–Computer Interaction* 5 (1990). https://doi.org/10.1080/07370024.1990.9667157
[6] Michael Muller. 2014. Curiosity, Creativity, and Surprise as Analytic Tools: Grounded Theory Method. In *Ways of Knowing in HCI*, Judith S. Olson and Wendy A. Kellogg (Eds.). 25–48. https://doi.org/10.1007/978-1-4939-0378-8_2
[7] 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 (2016). https://doi.org/10.1109/MC.2016.200
[8] Brad A. Myers, John F. Pane, and Amy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47 (2004). https://doi.org/10.1145/1015864.1015888
[9] Simon Thompson. 2004. Refactoring Functional Programs. In *Advanced Functional Programming (AFP).* https://doi.org/10.1007/11546382_9
[10] Bin Xie and Gavriel Salvendy. 2000. Review and reappraisal of modelling and predicting mental workload in single- and multi-task environments. *Work & Stress* 14 (2000). https://doi.org/10.1080/026783700417249