

# Theories of Programming

Thomas D. LaToza<sup>\*1</sup>, Amy Ko<sup>\*2</sup>, David C. Shepherd<sup>\*3</sup>,  
Dag Sjøberg<sup>\*4</sup>, and Benjamin Xie<sup>†5</sup>

1 George Mason University – Fairfax, US. [tlatoya@gmu.edu](mailto:tlatoya@gmu.edu)

2 University of Washington – Seattle, US. [ajko@uw.edu](mailto:ajko@uw.edu)

3 Virginia Commonwealth University – Richmond, US. [shepherdd@vcu.edu](mailto:shepherdd@vcu.edu)

4 University of Oslo, NO. [dagsj@ifi.uio.no](mailto:dagsj@ifi.uio.no)

5 University of Washington – Seattle, US. [bxie@uw.edu](mailto:bxie@uw.edu)

---

## Abstract

Much of computer science research focuses on techniques to make programming easier, better, less error prone, more powerful, and even more just. But rarely do we try to explain any of these challenges. Why is programming hard? Why is it slow? Why is it error prone? Why is it powerful? How does it do harm? These why and how questions are what motivated the Dagstuhl Seminar 22231 on Theories of Programming. This seminar brought together 28 CS researchers from domains most concerned with programming human and social activities: software engineering, programming languages, human-computer interaction, and computing education. Together, we sketched new theories of programming and considered the role of theories more broadly in programming.

**Seminar** June 6–10, 2022 – <http://www.dagstuhl.de/22231>

**2012 ACM Subject Classification** Social and professional topics → Computing education; Human-centered computing → Human computer interaction (HCI); Software and its engineering

**Keywords and phrases** computing education, human-computer interaction, programming languages, software engineering, theories of programming

**Digital Object Identifier** 10.4230/DagRep.12.6.1

## 1 Executive Summary

*Benjamin Xie (University of Washington – Seattle, US)*

*Amy Ko (University of Washington – Seattle, US)*

*Thomas D. LaToza (George Mason University – Fairfax, US)*

*David C. Shepherd (Virginia Commonwealth University – Richmond, US)*

*Dag Sjøberg (University of Oslo, NO)*

**License**  Creative Commons BY 4.0 International license

© Benjamin Xie, Amy Ko, Thomas D. LaToza, David C. Shepherd, and Dag Sjøberg

Mature scientific disciplines are characterized by their theories, synthesizing what is known about phenomena into forms which generate falsifiable predictions about the world. In computer science, the role of synthesizing ideas has largely been through formalisms that describe how programs compute. However, just as important are scientific theories about how programmers write these programs. For example, software engineering research has increasingly begun gathering data, through observations, surveys, interviews, and analysis of artifacts, about the nature of programming work and the challenges developers face, and evaluating novel programming tools through controlled experiments with software developers.

---

\* Editor / Organizer

† Editorial Assistant / Collector



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Theories of Programming, *Dagstuhl Reports*, Vol. 12, Issue 6, pp. 1–13

Editors: Thomas D. LaToza, Amy Ko, David C. Shepherd, and Dag Sjøberg



DAGSTUHL  
REPORTS

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

Computer science education and human-computer interaction research has done similar work, but for people with different levels of experience and ages learning to write programs. But data from such empirical studies is often left isolated, rather than combined into useful theories which explain all of the empirical results. This lack of theory makes it harder to predict in which contexts programming languages, tools, and pedagogy will actually help people successfully write and learn to create software.

Computer science needs scientific theories that synthesize what we believe to be true about programming and offer falsifiable predictions. Whether or not a theory is ultimately found to be consistent with evidence or discarded, theories offer a clear statement about our current understanding, helping us in prioritizing studies, generalizing study results from individual empirical results to more general understanding of phenomena, and offering the ability to design tools in ways that are consistent with current knowledge.

Dagstuhl Seminar 22231 on *Theories of Programming* explored the creation and synthesis of scientific theories which describe the relationship between developers and code within programming and social activities. The seminar brought together researchers from software engineering, human-computer interaction, programming languages, and computer science education to exchange ideas about potential theories of programming. We identified and proposed theories that arose from many sources: untested but strongly-held beliefs, anecdotal observations, assumptions deeply embedded in the design of languages and tools, reviews of empirical evidence on programming, and applications of existing theories from psychology and related areas. Our aim was to bridge this gulf: formulating deeply-held beliefs into theories which are empirically testable and synthesizing empirical findings in ways that make predictions about programming tools and languages.

To achieve this aim, the seminar had three specific goals. 1) Bring together researchers with diverse expertise to find shared understanding. 2) Create a body of theories which make testable predictions about the effects of programming tools, languages, and pedagogy on developer behavior in specific contexts. 3) Propose future activities which can advance the use of theories, including identifying studies to conduct to test theories and ways to use theories to communicate research findings to industry.

During this seminar, a few short talks first reviewed the nature, creation, and use of theories as well as existing evidence about developer behavior during programming activities. The main activity of the seminar was working in small groups to sketch new theories of programming.

## Seminar Overview

The seminar was divided into the following sessions across four days in June 2022:

- Tuesday: welcome, what is theory, describing theories, critiquing theories
- Wednesday: brainstorming unexplained programming phenomena, sketching theories, getting feedback on theories, and refining theories
- Thursday: presenting theory sketches, discussing ways of sharing theories, and skeptically examining whether developing theories of programming is really worth the time
- Friday: reflecting on takeaways and departure

The seminar was organized by Thomas Latoza, Amy J. Ko, Dag Sjøberg, David Shepherd, and Anita Sarma. Anita later had to drop out, leaving Thomas, Amy, Dag, and David as the four organizers who were able to attend.

## What is theory?

The goal of this opening session was to find common ground on what theory was. To achieve this, each organizer gave short presentations related to theories.

Thomas identified how researchers used theories to generate falsifiable predictions about the world. He described common characteristics of theories as abstract, explanatory, relevant, and operationalizable. An example of a theory of programming he provided was how violating constraints cause defects or reduces code quality.

Amy described an interpretivist framing of theories, where theories were cultural and experience-based. Some theories were folk theories (e.g. code is magic, Not Invented Here, and spaces vs tabs for white space). Some theories were personal, such as programming as common sense machines and tinkering towards correctness. Other theories came from research communities such as ICSE. For example, a theory of programming is that we can copy and adapt code from another location in a program to fix bugs.

Dag drew guidelines between what was and was not a theory. He identified multiple examples of what were not theories: scientific laws were not theories because they were missing the “why;” trivial statements were also not theories. The building blocks of theories included constructs, propositions, explanations, and scope. Theories can help us explain surprising empirical results, while empirical results can help us support or refute certain theories. Finally, Dag noted how premature theorizing is likely to be wrong, but can still be useful.

David emphasized the importance of keeping theories practical. He defined a relationship from theorems to corollaries to examples and applications. He provided an example of using different representations in music for different use cases and users.

In open discussions and breakout groups, attendees identified additional nuances to theories. We noted how it is useful for theories to enable ease of communication or shared understanding. But by defining a vocabulary, theories can also limit the scope of explanation. We can also use theories to understand what we observe or to justify interventions. Finally, there was discussion about creating theories inductively, deductively, and/or abductively.

Common themes that arose from discussion include how theories are seldom used to justify the design of programming languages and tools, and how programming is a social endeavor and drawing upon social science research (e.g. psychology) can support theory building.

## Expressing Theories using a Theory Template

The goal of this session was to try to express theories using a theory template developed by the organizers. While the goal of this template was to support the creation of new theories, attendees used it to describe existing theories for this session. Attendees broke into five groups to attempt to apply the theory template to the following existing theories of programming:

- Asking and answering questions [1]
- Program comprehension as fact finding [2]
- Leaky abstractions [3]
- Information hiding [4]
- Theory of programming instruction [5]

After considering feedback from attendees, organizers revised the theory template. The revised theory template’s section headers and helper text are as follows:

1. *Theory's name*: Choose a name that is memorable, short, and descriptive.
2. *Summary*: In a few sentences, summarize the phenomena, constructs, relationships, and a concrete example, hypothesis, and study.
3. *Contributors*: Who has contributed to this theory? Add your name here.
4. *Phenomena*: What programming phenomena is your theory trying to explain? And in what scope (people, expertise, contexts, tools, etc.)? This description should just describe what is being explained, should not offer an explanation; that below. (“Programming” includes any and all interactions between people and code, in any context (e.g., software engineering, learning, play, productivity, science, and all of the activities involved in creating programs, including requirements, architecture, implementation, verification, monitoring, and more).
5. *Prior Work*: What prior work offers an explanation of this phenomena, or might help generate an explanation of this phenomena? For the purposes of the seminar, this does not need to be complete, but a complete description of this theory would have an extensive literature review covering theories that inspired this theory, as well as conflicting theories.
6. *Concepts*: Describe the key concepts of the theory and some concrete examples of them, building upon the phenomena above. These might be variables, processes, people, aspects of people, structures, contexts or other phenomena that are essential to the theory's account of the phenomena. Note: concepts should be descriptions of ideas that give some structure and precision to describing the phenomena, not operationalizations or measurements – those belong in example hypotheses and/or studies.
7. *Relationships and Mechanisms*: Using the constructs described above, explain the causality of how the phenomena works. What causes what and how? Provide a few concrete examples to illustrate the idea.
8. *Example Hypotheses*: What testable claims do the constructs, relationships, and mechanisms imply?
9. *Example Studies*: What are existing or envisioned example study methods that might investigate the hypotheses above? How might the concepts be operationalized and measured? Describe details about populations, samples, tasks, contexts, tools, observations. Remember that studies can involve many forms of observation and data, both qualitative and quantitative and even design contributions. Studies do not have to be feasible to be proposed and can vary in scope, from single-study sized methods to long-term research agendas that might explore a theory over many years and many projects.
10. *Corollaries*: What follows from this theory, if true? Provide potential implications, concrete or otherwise.

## Unexplained phenomenon

After spending the first day discussing what theories were and applying a theory template, the goal of the second day was to identify unexplained phenomena related to programming and apply theories to explain them. After an informal voting process, attendees created groups to develop theories around the following phenomena:

- Debugging
- Types
- Neurodiversity in programming
- Data programming
- Code examples

- Developer tools
- Learning effects from code analysis

Groups spent all of Wednesday developing theories by filling out the theory template and then getting feedback from members of other groups. They then iterated and created presentations. See included abstracts of talks for descriptions of each presentation.

## Sharing Theories

On Thursday after the presentations, attendees had discussions about how to share theories of programming to broader audiences. Many ideas included written dissemination, such as publishing research, writing books, creating a wiki, adding to reviewer guidelines, creating a website, defining syllabi for reading groups, speaking on podcasts, and posting on social media sites. Other ideas featured opportunities for further interaction, such as workshops, special interest groups, demonstrations of theories for practitioners, and stickers/flair for engagement at conferences. Other ideas focused on incentive structures, such as creating a “best new theory” award at conferences.

Group-wide discussions about sharing theories identified some structural barriers and opportunities. A barrier to broader theory creation and/or use is that most computing researchers do not have much training in theories. Workshops, reading groups, or changes to undergraduate or graduate level coursework could help address this. Another structural barrier is that most conferences lack instructions about theory. Adding instructions in paper calls and reviewer instructions as well as “theory shepherds” could help address this systemic barrier.

## Do we really need theories?

The final session for Thursday was critically reflective about whether programming actually required theories. Given this session occurred after lunch on the final full day, this session got silly. After splitting into groups to discuss, groups shared eclectic presentations to reflect their discussions:

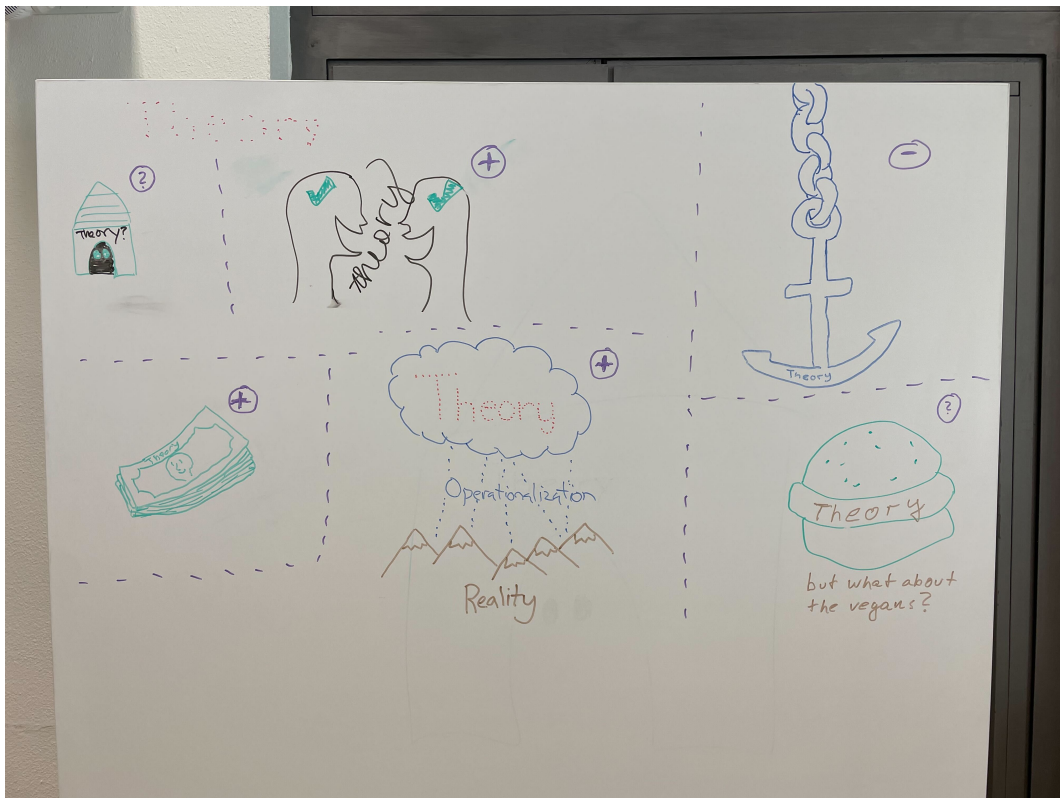
- a colorful whiteboard diagram about pros and cons of theories (Fig. 1)
- Another whiteboard diagram about whether to use theory (Fig. 2)
- A list of bullets about challenges of making changes in publishing
- An humorous improv skit about a conference Q&A session on a theory paper.

## Reflections on the week

The final session of this seminar asked attendees to reflect on the seminar as a whole. Attendees identified some high-level takeaways:

Attendees found theories useful for helping understand why things (e.g. languages or tools) do or do not work. They also found theories helpful for differentiating between how we think people work and how they actually work.

Attendees also felt that the engagement of computing researchers with theories of programming was often limited by the lack of interest and/or lack of expertise. Interdisciplinary



■ **Figure 1** Whiteboard sketches of the pros and cons of theories, as depicted by various diagrams.

research can help create the gestalt of expertise required to create theories of programming, but narrow conference and journal scopes often make this difficult. Specifically, many computing researchers lack expertise in empirical evaluations, making it difficult to develop rigorous evidence that is often foundational to theory building. Furthermore, much training in empirical evaluations focuses on lab settings, whereas most programming happens “in the wild.”

Multiple attendees also felt that theories were more implicitly prevalent in computing research than was explicitly discussed. Some conversation focused on “lower case ‘t’” theories, or theories that we not fully formalized, but provided use and explanation. Many attendees felt that theories implicitly existing in papers, but were unaware of explanations into this work.

A concluding consensus was that theories of programming have existed in the background. Through explicit engagement and discourse, this Dagstuhl Seminar could serve as a catalyst to augment existing theories and craft new ones.

## References

- 1 Sillito, Jonathan, Gail C. Murphy, and Kris De Volder. “Asking and Answering Questions during a Programming Change Task.” *IEEE Transactions on Software Engineering* 34, no. 4 (July 2008): 434–51. <https://doi.org/10.1109/TSE.2008.26>.
- 2 LaToza, Thomas D., David Garlan, James D. Herbsleb, and Brad A. Myers. “Program Comprehension as Fact Finding.” In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations*



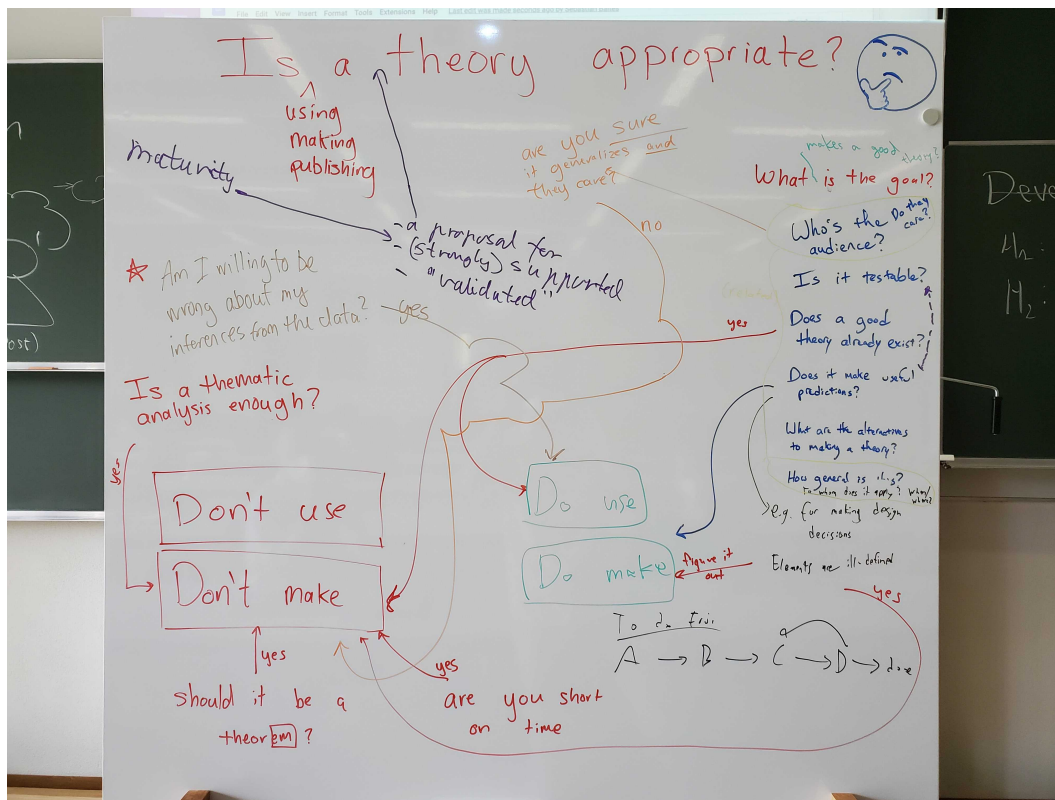


Figure 2 Whiteboard sketch of flowchart considering whether a theory is appropriate.

of Software Engineering, 361–70. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007. <https://doi.org/10.1145/1287624.1287675>.

- Spolsky, Joel. "The Law of Leaky Abstractions." Joel on Software, November 11, 2002. <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/>.
- Parnas, David L. "On the criteria to be used in decomposing systems into modules." *Pioneers and their contributions to software engineering*. Springer, Berlin, Heidelberg, 1972. 479-498.
- Xie, Benjamin, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. "A Theory of Instruction for Introductory Programming Skills." *Computer Science Education*, January 25, 2019, 1–49. <https://doi.org/10.1080/08993408.2019.1565235>.

## 2 Table of Contents

### Executive Summary

*Benjamin Xie, Amy Ko, Thomas D. LaToza, David C. Shepherd, and Dag Sjøberg* 1

### Overview of Talks

Formulating and Checking Hypotheses in Debugging

*Moritz Beller, Sebastian Balthes, Jonathan Bell, Brittany Johnson-Matthews, and Hila Peleg* . . . . . 9

Tool-Tainted Knowledge Guides Developer Decisions (TTKGDD)

*Thomas Fritz, Tudor Girba, Gail C. Murphy, Dag Sjøberg, and Kathryn T. Stolee* . 9

Theory of Code Examples

*Jun Kato, Gunnar Bergersen, Scott Fleming, Robert Hirschfeld, and Andreas Zeller* 10

Types as tools for structuring thought

*Sarah Lim, Michael Coblenz, Andrew Head, and Thomas D. LaToza* . . . . . 11

Learning Effects from Code Analysis

*Justin Lubin, Francisco Servant, Justin Smith, and Emma Söderberg* . . . . . 11

Neurofriction in Programming Tools

*Jeffrey Stylos, Amy Ko, and Lutz Prechelt* . . . . . 12

Narrative Data Programming: Narrative First, Program Second

*Benjamin Xie and David C. Shepherd* . . . . . 12

**Participants** . . . . . 13



## 3 Overview of Talks

### 3.1 Formulating and Checking Hypotheses in Debugging

*Moritz Beller (Facebook – Menlo Park, US), Sebastian Baltes (University of Adelaide, AU), Jonathan Bell (Northeastern University – Boston, US), Brittany Johnson-Matthews (George Mason University – Fairfax, US), and Hila Peleg (Technion – Haifa, IL)*

**License** © Creative Commons BY 4.0 International license  
© Moritz Beller, Sebastian Baltes, Jonathan Bell, Brittany Johnson-Matthews, and Hila Peleg

Particularly in large software systems, complex bugs may entirely stump developers who attempt to debug them solely through breakpoints and printf statements. One strategy for debugging failing test cases is “scientific debugging” – the process of formulating hypotheses that could explain the failure, and then investigating the code and its execution to see which hypotheses hold and refine them. Some programmers generate more accurate hypotheses than others; some programmers are more efficient at prioritizing and checking hypotheses than others; some may even intuitively jump from a failing test to the valid hypothesis. We have formulated a theory of scientific debugging to enable the externalization of the process that experts follow when debugging, providing a framework for future research in the growth and transfer of expertise. In our theory, developers use experiences from previous debugging to sort through patterns of symptoms and related hypotheses, which helps them navigate the hypothesis space. They iteratively select new working hypotheses that when checked either hold or are refuted, providing more information to diagnose the underlying root cause of the failure. Developers might use different strategies to navigate throughout the hypothesis space, likely without tool support.

### 3.2 Tool-Tainted Knowledge Guides Developer Decisions (TTKGDD)

*Thomas Fritz (Universität Zürich, CH), Tudor Girba (feenk – Wabern, CH), Gail C. Murphy (University of British Columbia – Vancouver, CA), Dag Sjøberg (University of Oslo, NO), and Kathryn T. Stolee (North Carolina State University – Raleigh, US)*

**License** © Creative Commons BY 4.0 International license  
© Thomas Fritz, Tudor Girba, Gail C. Murphy, Dag Sjøberg, and Kathryn T. Stolee

During the “Theories of Programming” seminar, we developed an initial version of a theory entitled “Tool-Tainted Knowledge Guides Developer Decisions (TTKGDD)”. This theory addresses observations that have been made about programming today, particularly that all too often, programmers make decisions about their program based on beliefs rather than evidence. Our theory is that basing decisions on evidence requires contextual tools that extract and present facts about the system in terms that the programmer can easily comprehend. Taking an evidence-based approach leads to higher quality decisions being made about the system.

The major concepts in the theory are developers, tools, decisions, code and hypotheses. There are many relationships between these concepts, such as how developers form hypotheses about their code and how developer’s knowledge guides the choice of a tool to answer a hypothesis. This theory suggests many hypotheses to test, including that “contextualized tools lead to better developer decisions”, “too much automation in tools reduces knowledge of the system” and “biased tools lead to biased knowledge and therefore biased decisions”.

Interestingly, with this last hypothesis, the choice of the term “biased” may indeed bias experiments conducted. Equally interesting hypotheses could be “opinionated tools lead to opinionated knowledge and therefore opinionated decisions” or “appropriate tools lead to appropriate knowledge and therefore appropriate decisions”.

We intend to propose a workshop at a conference that investigates the meaning of a “contextual tool”, the implications of applying them in concrete scenarios, and refinements of the theory.

### 3.3 Theory of Code Examples

*Jun Kato (AIST – Tsukuba, JP), Gunnar Bergersen (University of Oslo, NO), Scott Fleming (University of Memphis, US), Robert Hirschfeld (Hasso-Plattner-Institut, Universität Potsdam, DE), and Andreas Zeller (CISPA – Saarbrücken, DE)*

License © Creative Commons BY 4.0 International license  
© Jun Kato, Gunnar Bergersen, Scott Fleming, Robert Hirschfeld, and Andreas Zeller

We were concerned with concrete examples of both code and data in programming activities. Often such examples come in small sizes, as collections rather than single snippets, and are generally considered beneficial to help people understand and extend a codebase or system. Despite this common understanding, examples take many different forms, and there remain numerous challenges to ensuring their benefits.

The authors come from multiple disciplines including Human-Computer Interaction, Software Engineering, Computer Science Education, and Programming Languages, and use the same terminology “code examples” mainly in the following contexts.

First, there are code examples in tutorials and learning materials for computer science education. Several key issues that arise in the use of examples for teaching and learning. Authoring educational examples holds significant challenges in terms of producing correct (e.g., well-tested) code examples and of keeping examples up to date in the face of rapidly evolving platforms and APIs. Designing effective worked examples (i.e., which entail a problem statement, solution steps, and the final solution to the problem) similarly holds challenges with respect to helping learners gain transferable problem-solving knowledge and an understanding of the rationale that underlies the solution steps.

Second, there are code examples in exploratory programming. Exploratory programmers have an open-ended goal, learning about a new domain, working toward a specification and growing a system. To make sure their code works correctly and to find an appropriate next step for their exploration, they provide multiple examples and examine the results of their execution. The major challenge is the difficulty of writing good code examples that cover the test cases of current interest, run in a reasonable time frame, and provide informative feedback for the next steps. We foresee that addressing these issues will require further efforts on improving the liveness of the programming environment and adding guidance based on code understanding techniques, including static and dynamic code analysis.

While we saw differences in these contexts such as the “goodness” criteria for the code examples, we also found similarities like the need to support the authoring process of good code examples. Possible areas of study include how to keep examples relevant to the purpose, how to organize examples in the order that makes the most sense to the learners and programmers, and how to make examples more informative and explorable.

### 3.4 Types as tools for structuring thought

*Sarah Lim (University of California – Berkeley, US), Michael Coblenz (University of Maryland – College Park, US), Andrew Head (University of Pennsylvania – Philadelphia, US), and Thomas D. LaToza (George Mason University – Fairfax, US)*

License © Creative Commons BY 4.0 International license  
© Sarah Lim, Michael Coblenz, Andrew Head, and Thomas D. LaToza

Existing theories about types focus mainly on formal semantics, without considering how type systems actually influence the human practice of programming. Our theory aims to characterize how static type systems can shape the user experience of programming beyond simply surfacing type errors. In our theory, one key way that types can support programmers is by helping a programmer encode an ontology of the domain while planning their program, which will later support reasoning in terms of domain-specific constructs. Ultimately, this leads a type system to support a programmer during ideation, solution search, refactoring, and program comprehension. This encoding can be promoted or inhibited according to the features of the type system in which the work is done, and the expressivity of the type system affects programmers' success in encoding relevant constraints. Then, when programmers use the resulting encoding, their search for an appropriate solution can be guided or inhibited by the constraints in the encoding.

We theorize that well-designed types within a sufficiently expressive type system can (1) catch common mistakes and offload verification work to the computer; (2) help programmers identify good solutions to their problems; and (3) allow types to be expressed in a way that matches the problem domain in the way the programmer thinks about it. Importantly, rich type systems provide counterparts to the execution-based strategies common in dynamically-typed languages, such as defining example data, watching unit tests, or working heavily with a REPL during implementation. We propose future research to explore which design decisions around types support programmers in the tasks described above.

### 3.5 Learning Effects from Code Analysis

*Justin Lubin (University of California – Berkeley, US), Francisco Servant (King Juan Carlos University – Madrid, ES), Justin Smith (Lafayette College – Easton, US), and Emma Söderberg (Lund University, SE)*


License © Creative Commons BY 4.0 International license  
© Justin Lubin, Francisco Servant, Justin Smith, and Emma Söderberg

Developers typically use code analysis tools to improve code quality. We posit that these tools have an equally transformative impact on developers' mental models of a problem domain. For instance, reachability analysis tools may reveal incorrect models of possible states in a system, lifetime analysis in Rust may prompt developers to decide how long certain objects should live in their models, and dependency analysis may reveal circular reasoning in developers' mental models. We theorize that the extent to which the mental model of the problem domain and the embodiment in the code base and by extension the code analysis tools overlap affects the extent to which domain-specific learning effects can occur.

We spent several sessions mapping out an initial framework about the concepts, processes, relationships involved in code analysis tools affecting the mental model of developers as they write code. These discussions resulted in the formation of a team of researchers who will explore this topic further and in a plan to formalize and disseminate our findings in future publications.

### 3.6 Neurofriction in Programming Tools

*Jeffrey Stylos (Stylos Research – Northampton, US), Amy Ko (University of Washington – Seattle, US), and Lutz Prechelt (FU Berlin, DE)*

License  Creative Commons BY 4.0 International license  
© Jeffrey Stylos, Amy Ko, and Lutz Prechelt

Some people approach programming systematically, making plans and then implementing. Other people approach programming more opportunistically, working through examples and then building programs as they test and gather feedback. Others still may have different approaches to programming, shaped by how they learn, process information, and manage their attention.

Our ecosystem of tools, languages, and APIs are mostly created by people who are more systematic, disadvantaging those who do prefer to be more opportunistic, or have other problem solving approaches and preferences. This creates a mismatch between tools and programmers’ needs that produces what we call “Neurofriction”. Some of this friction is even framed as desirable by tool designers, describing some ways of programming as the “right” or “desirable” way, stigmatizing other ways of working. This is complicated by collaboration, where one team may need to agree on a particular set of languages, tools, and processes that further create Neurofriction.

By understanding Neurofriction better, and developing understanding amongst tool, language, and API designers about diverse needs, we may be able to create more universal tool designs.

### 3.7 Narrative Data Programming: Narrative First, Program Second

*Benjamin Xie (University of Washington – Seattle, US) and David C. Shepherd (Virginia Commonwealth University – Richmond, US)*

License  Creative Commons BY 4.0 International license  
© Benjamin Xie and David C. Shepherd

Computational notebooks are often criticized for their lack of adherence to traditional software engineering best practices. While this is certainly true, and often problematic, there may be good reasons for this departure from accepted norms. Because their purpose is to tell a story, we believe that computational notebooks should be seen as narratives first, and programs second. That is, while essential qualities like correctness are still important, the narrative that is woven from top to bottom of the notebook should take precedence over other non-essential qualities, such as efficiency and code reuse. Viewing notebooks in this way will allow us, as a community, to properly support users with essential best practices without over-burdening these often novice and end-user programmers with unnecessary complexity from practices, tools, and environments.

## Participants

- Sebastian Baltes  
University of Adelaide, AU
- Jonathan Bell  
Northeastern University –  
Boston, US
- Moritz Beller  
Facebook – Menlo Park, US
- Gunnar Bergersen  
University of Oslo, NO
- Michael Coblenz  
University of Maryland –  
College Park, US
- Scott Fleming  
University of Memphis, US
- Thomas Fritz  
Universität Zürich, CH
- Tudor Girba  
feenk – Wabern, CH
- Andrew Head  
University of Pennsylvania –  
Philadelphia, US
- Robert Hirschfeld  
Hasso-Plattner-Institut,  
Universität Potsdam, DE
- Brittany Johnson-Matthews  
George Mason University –  
Fairfax, US
- Jun Kato  
AIST – Tsukuba, JP
- Amy Ko  
University of Washington –  
Seattle, US
- Thomas D. LaToza  
George Mason University –  
Fairfax, US
- Sarah Lim  
University of California –  
Berkeley, US
- Justin Lubin  
University of California –  
Berkeley, US
- Gail C. Murphy  
University of British Columbia –  
Vancouver, CA
- Hila Peleg  
Technion – Haifa, IL
- Lutz Prechelt  
FU Berlin, DE
- Francisco Servant  
King Juan Carlos University –  
Madrid, ES
- David C. Shepherd  
Virginia Commonwealth  
University – Richmond, US
- Dag Sjøberg  
University of Oslo, NO
- Justin Smith  
Lafayette College – Easton, US
- Emma Söderberg  
Lund University, SE
- Kathryn T. Stolee  
North Carolina State University –  
Raleigh, US
- Jeffrey Stylos  
Stylos Research –  
Northampton, US
- Benjamin Xie  
University of Washington –  
Seattle, US
- Andreas Zeller  
CISPA – Saarbrücken, DE

