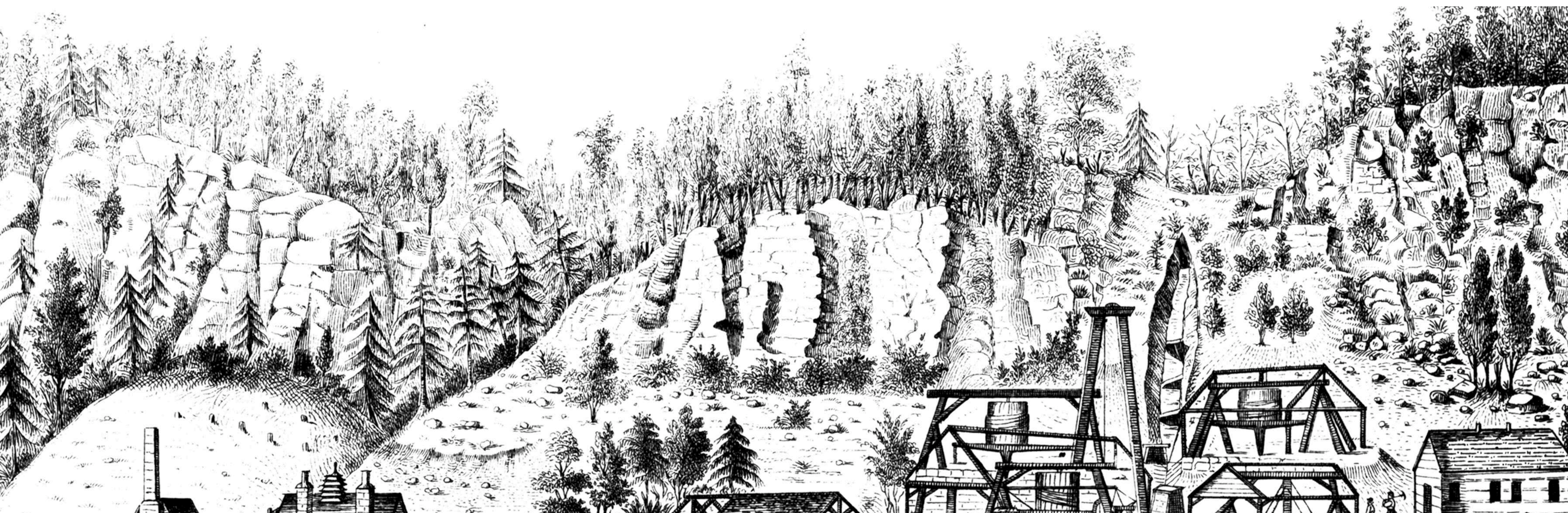
Mining the Mind, Minding the Mine

Grand Challenges in Comprehension and Mining

Andy J. Ko, Ph.D.

UNIVERSITY of WASHINGTON DUB DESIGN USE BUILD PLSE



Inter•disciplin•arity

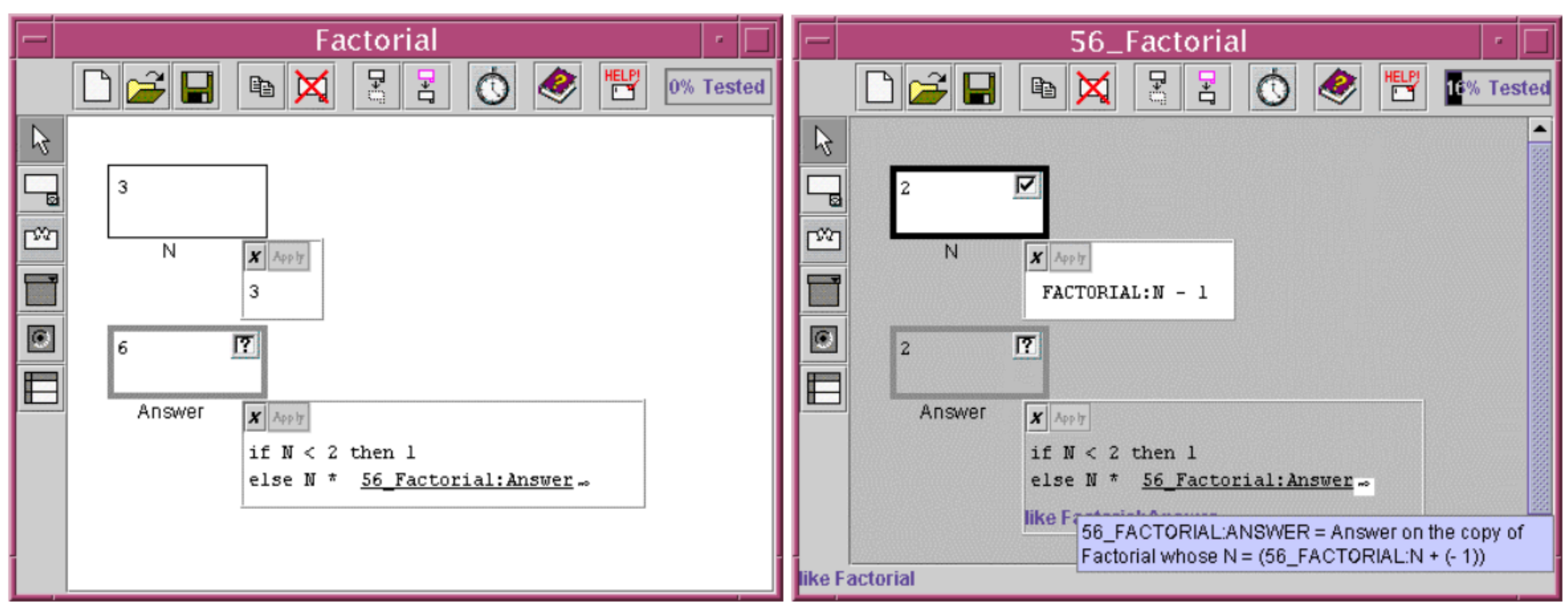
Drawing upon two or more branches of knowledge

About me

- Associate Professor at the UW Information School
- Background in CS, psychology, design, learning
- I **study** and **invent** interactions with code
- I **theorize** about what *programming* is
- I do all of this work at the boundaries between disciplines

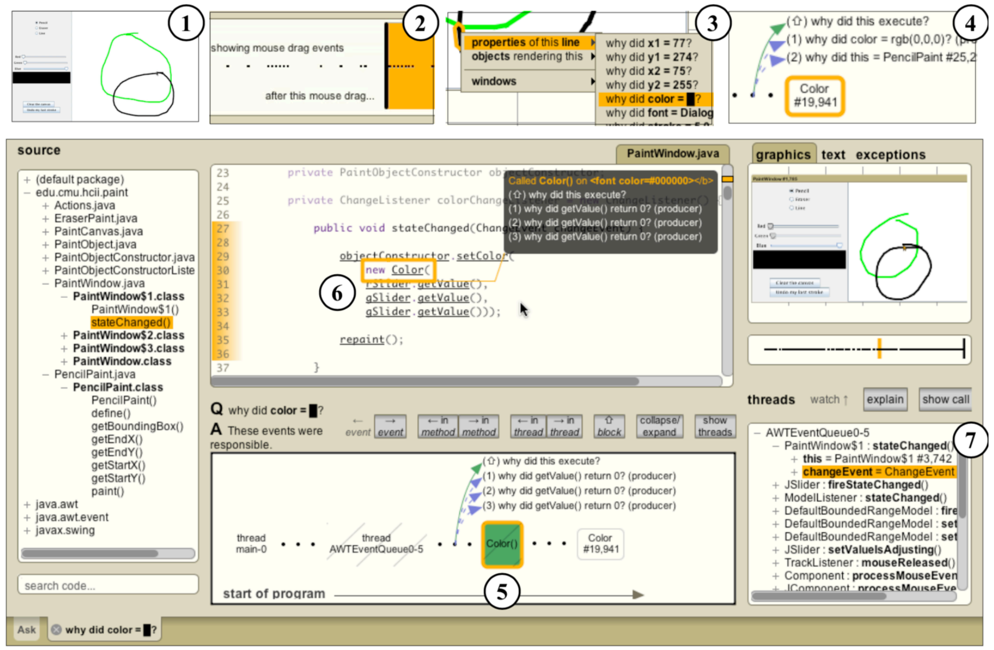
1999-2002 undergrad

- Worked with Margaret Burnett
- End-user programmers + spreadsheets
- How do we help end users test effectively *without any testing skills*?



2002–2008 Ph.D.

- Worked with Brad Myers at Carnegie Mellon
- How can we make debugging easier, faster using methods from human-computer interaction?

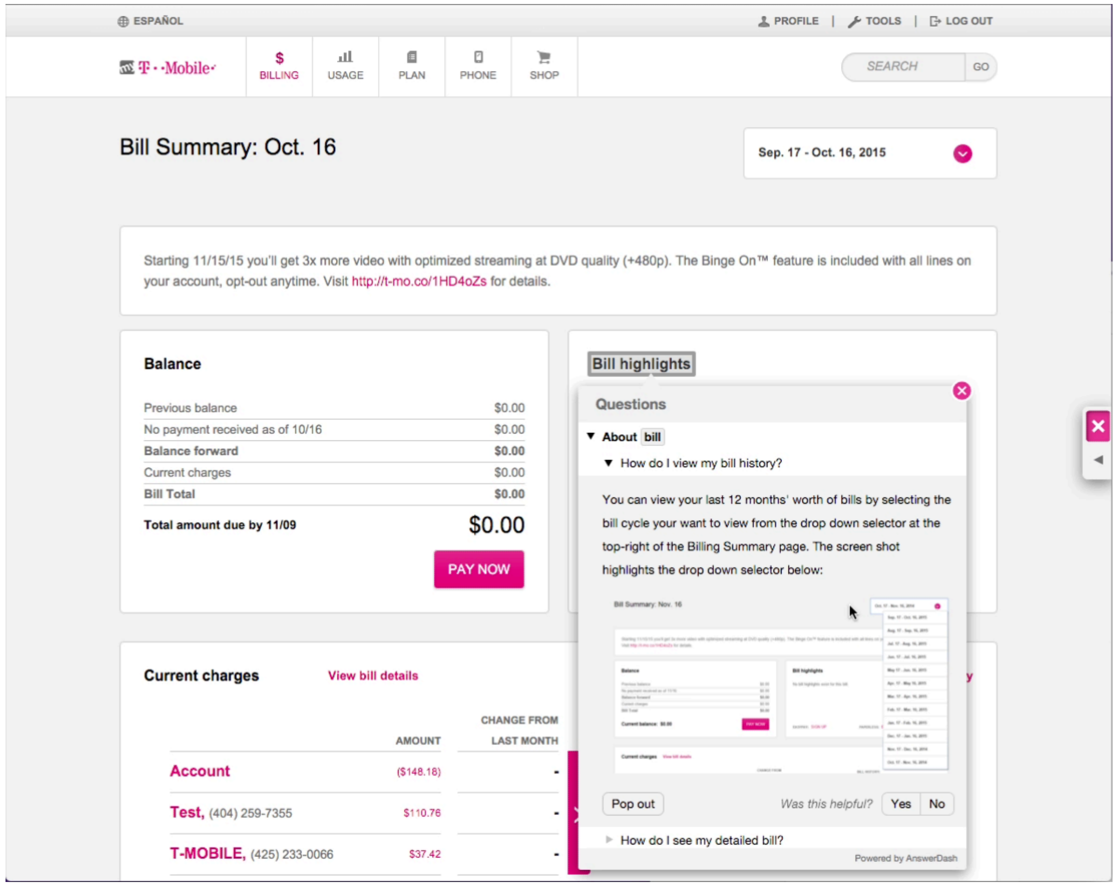


The Whyline

Come to my Most Influential Paper award talk at ICSE on Friday

2008-2014 pre-tenure

- University of Washington **Information School** (plus 4 years at AnswerDash, a startup I co-founded)
- How can we discover field failures at scale?
- How can we make bug triage *evidence-based*?



2014-present post-tenure

- Better software through *better developers*
  - Learning to code at scale
  - Rapid PL+API learning
  - Software engineering expertise

My history with comprehension and mining

- I’ve studied program comprehension since 1999, attended my first IWPC in 2003 (Portland, OR, USA)
- I’ve mined software repositories since 2005 when I downloaded my first dump of the Linux, Apache, and Firefox bug repositories
- But…I haven’t attended ICPC for 15 years and haven’t *ever* attended MSR!
- Unique opportunity for me to reflect as an outsider

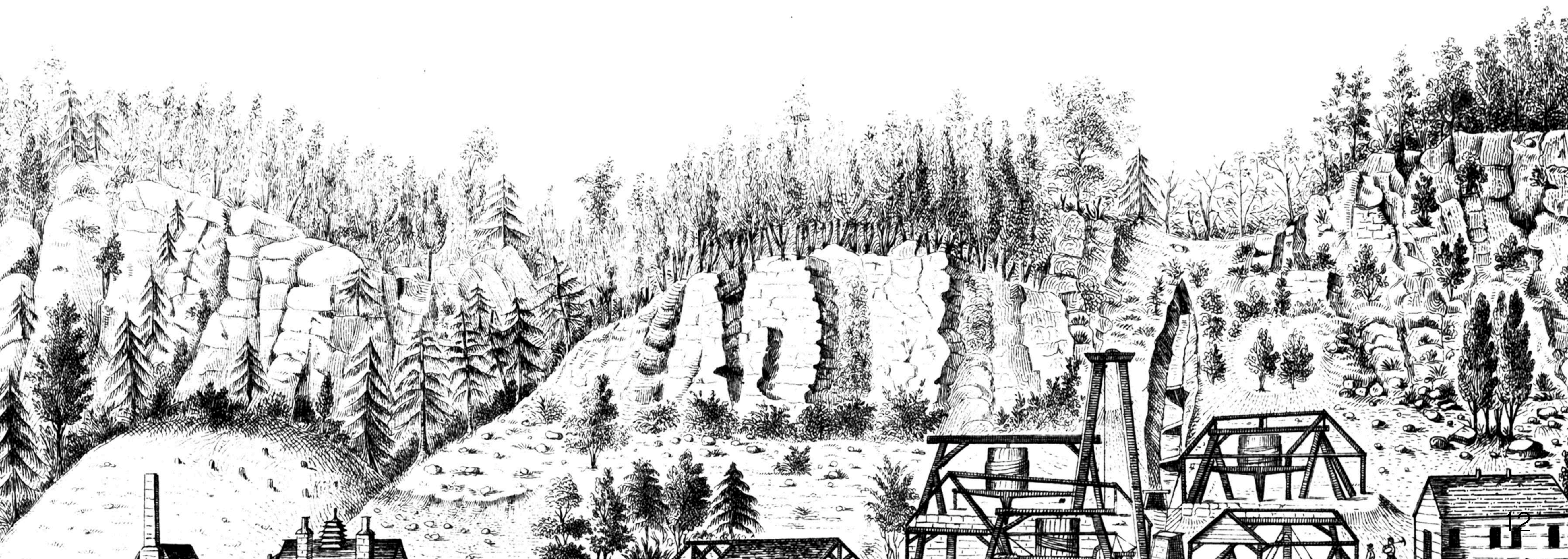
Who here regularly attends *ICPC*?

Who here regularly attends *MSR*?

Who here regularly attends *both*?

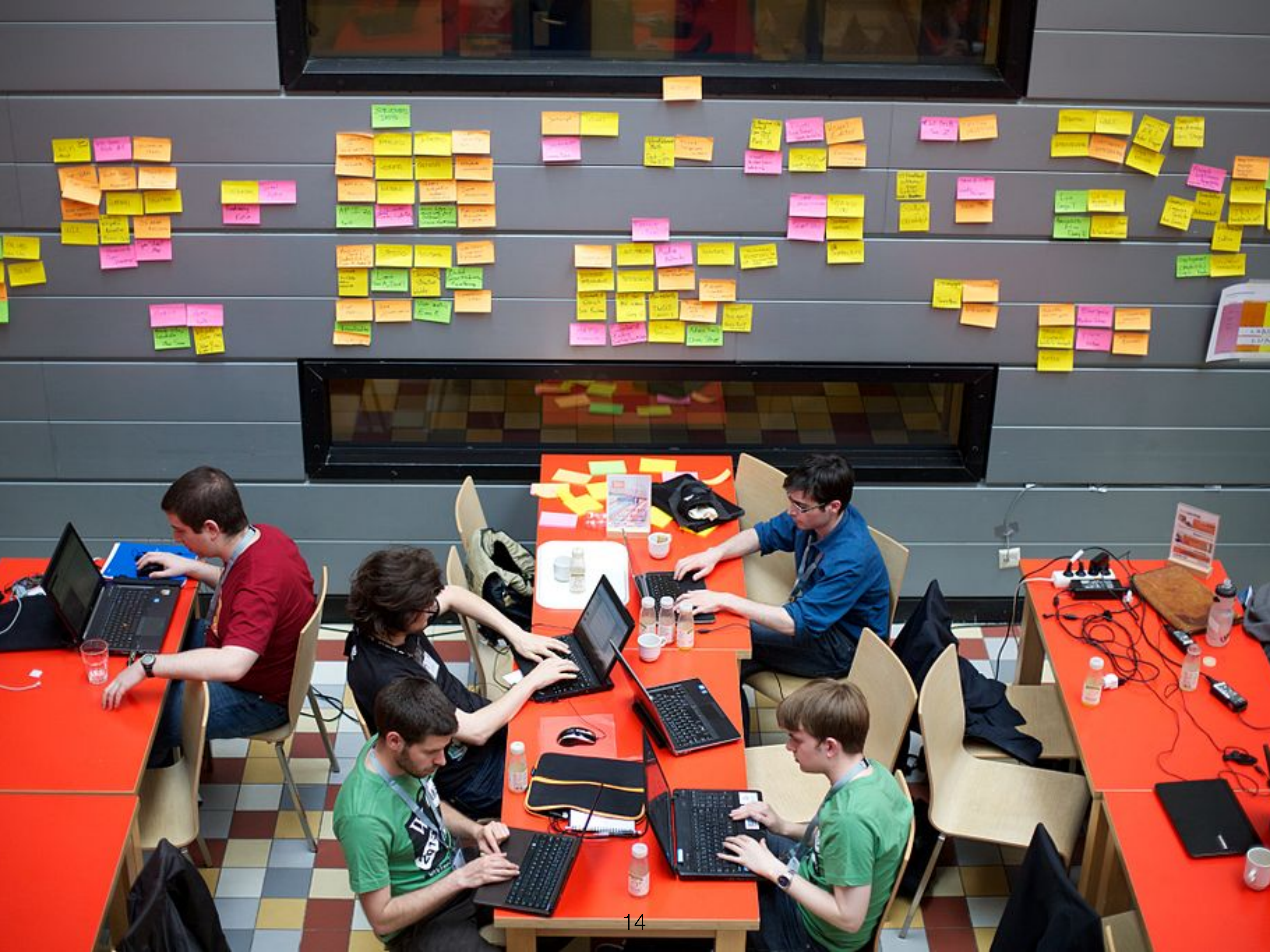
This talk

- How I see the MSR and ICPC communities
- Four missed opportunities at their intersection
- Next steps



Disclaimer

- In attempting to build a bridge between these communities, I'm going to identify *weaknesses* in each community
- Please don't take it personally; my work has the same weaknesses.
- Everyone here is doing great work, but to make it even greater, we must surface our disciplinary shortcomings.

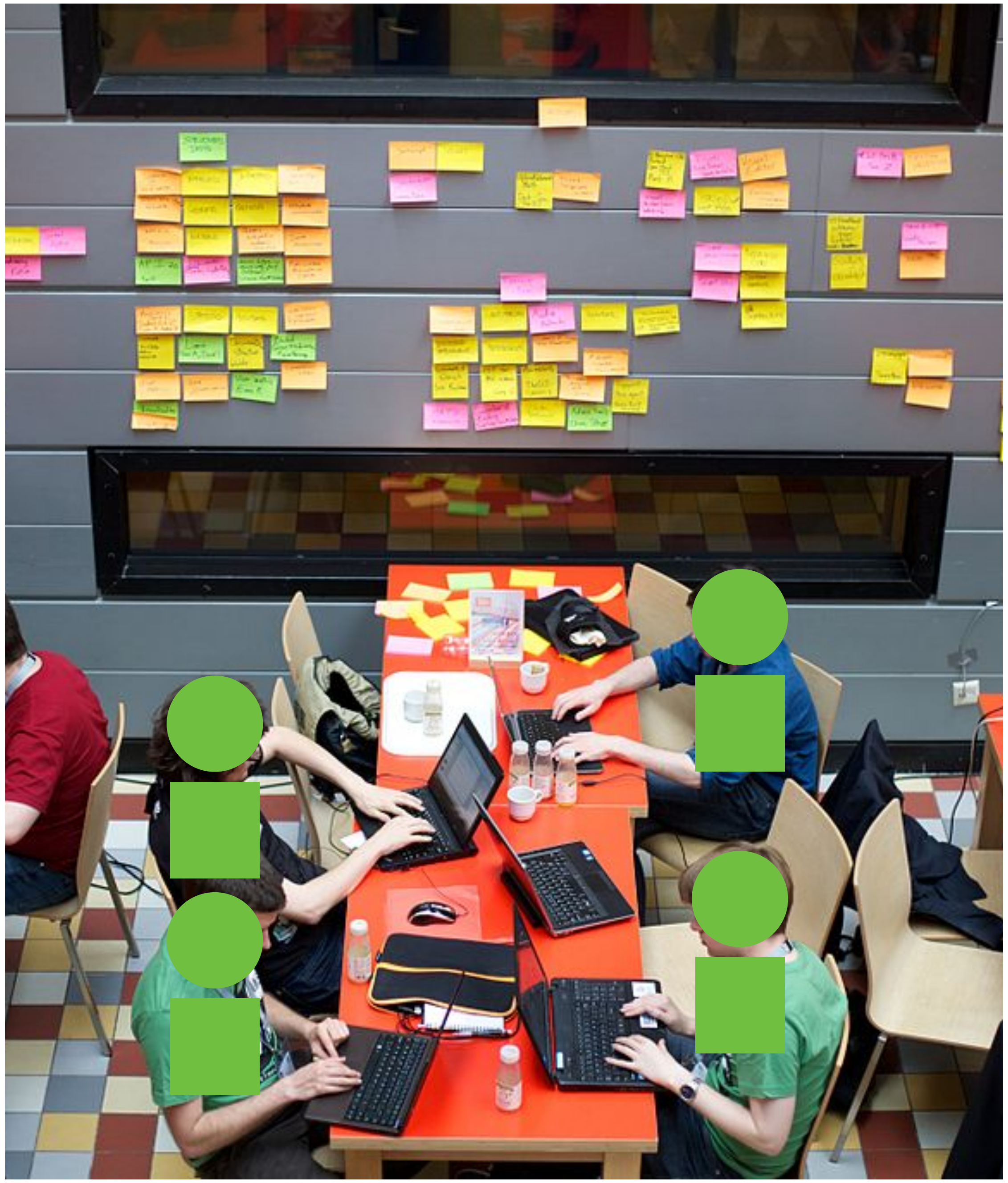


What we have in common

- *All* of us want to making programming and software engineering more effective, efficient, enjoyable, and successful
- *All* of us want to do this through rigorously discovery, of new tools, processes, insights
- We only differ in *how* we do this research (methods), and what we believe will make a difference (phenomena)

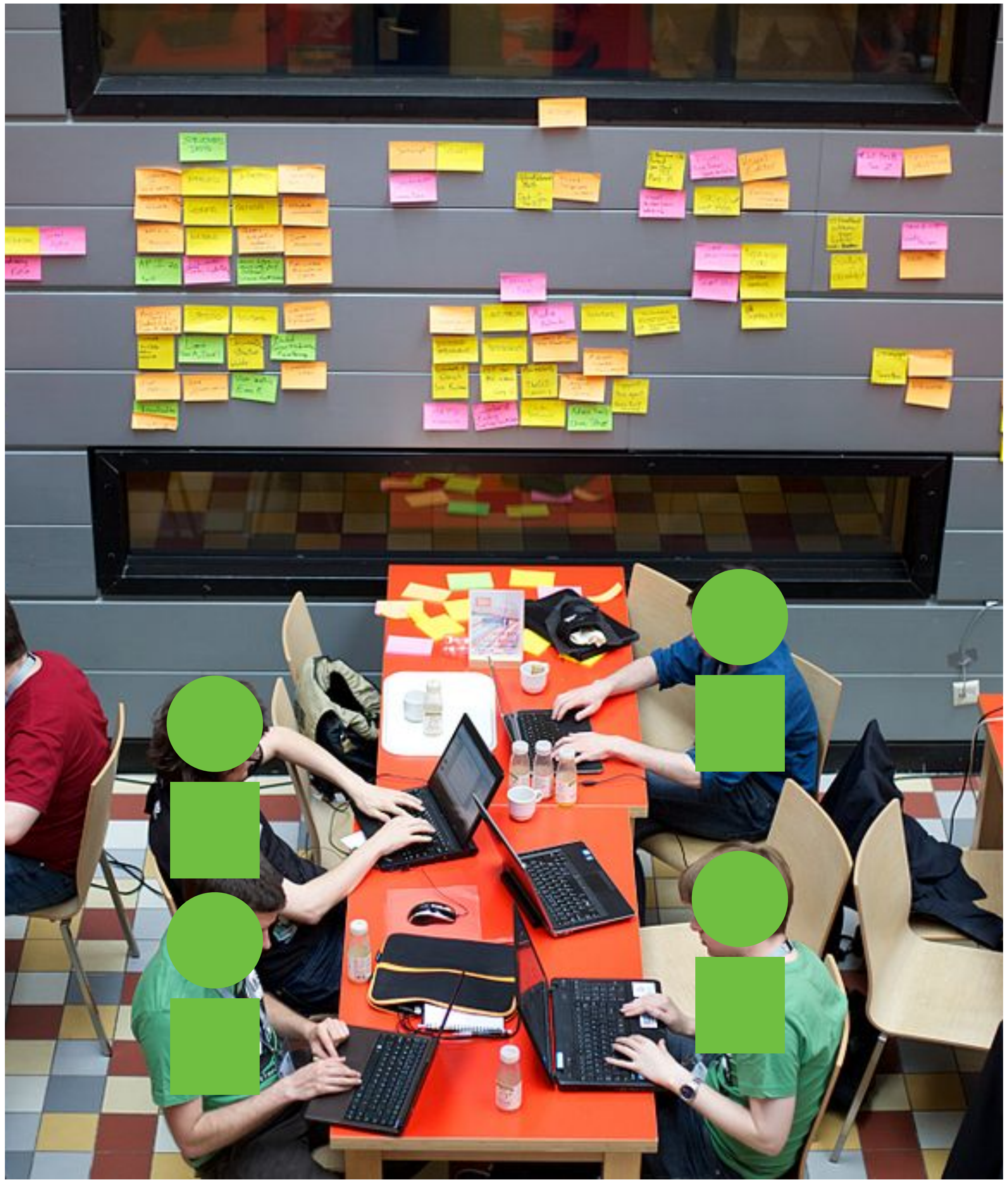
Comprehension

- Units of analysis
  - Perception
  - Cognition
  - Decisions
  - Collaboration
  - Contexts

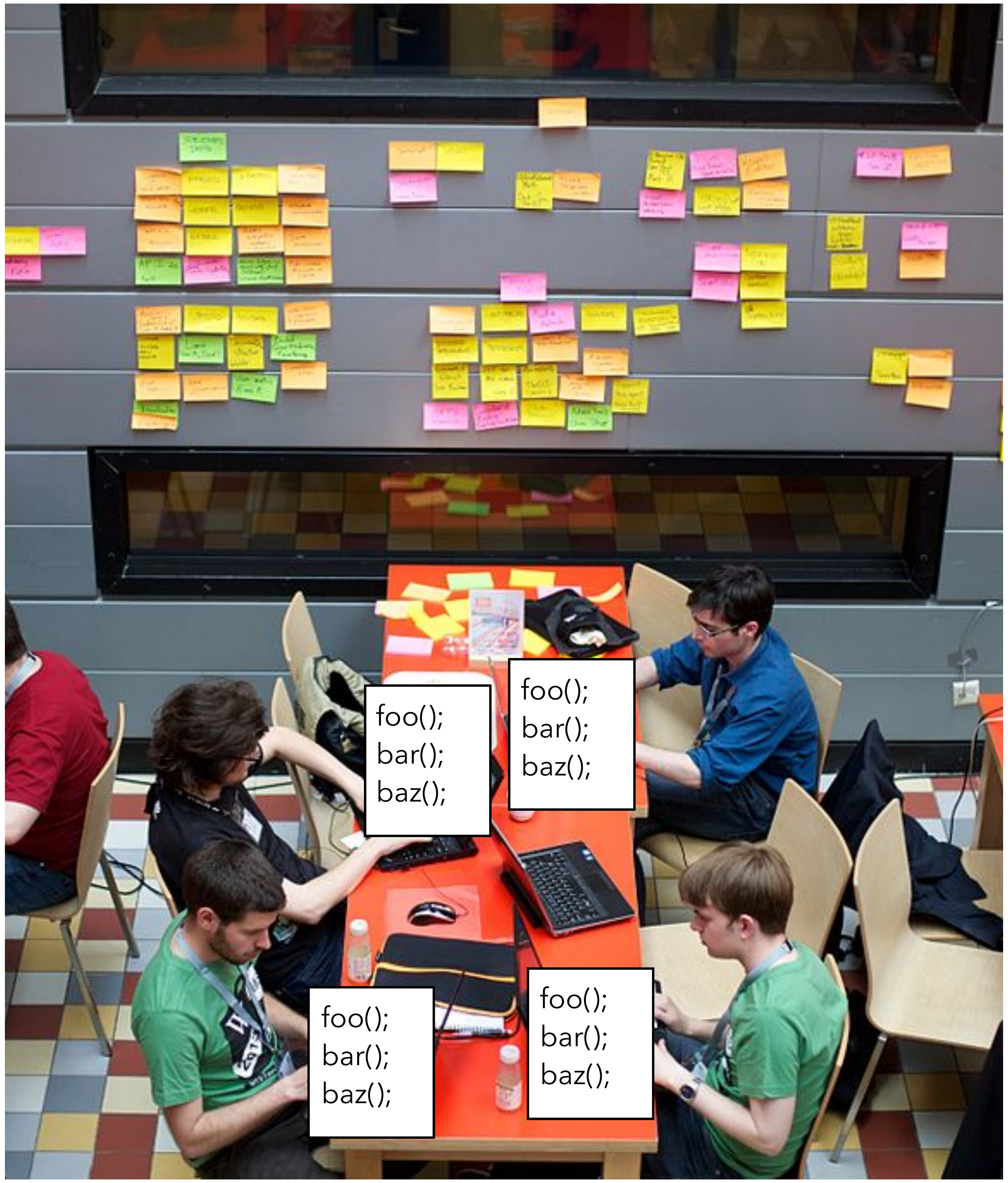


Comprehension

- New science on human program comprehension
- New tools to support developer's program comprehension
- Evaluations of strengths and weaknesses of comprehension tools

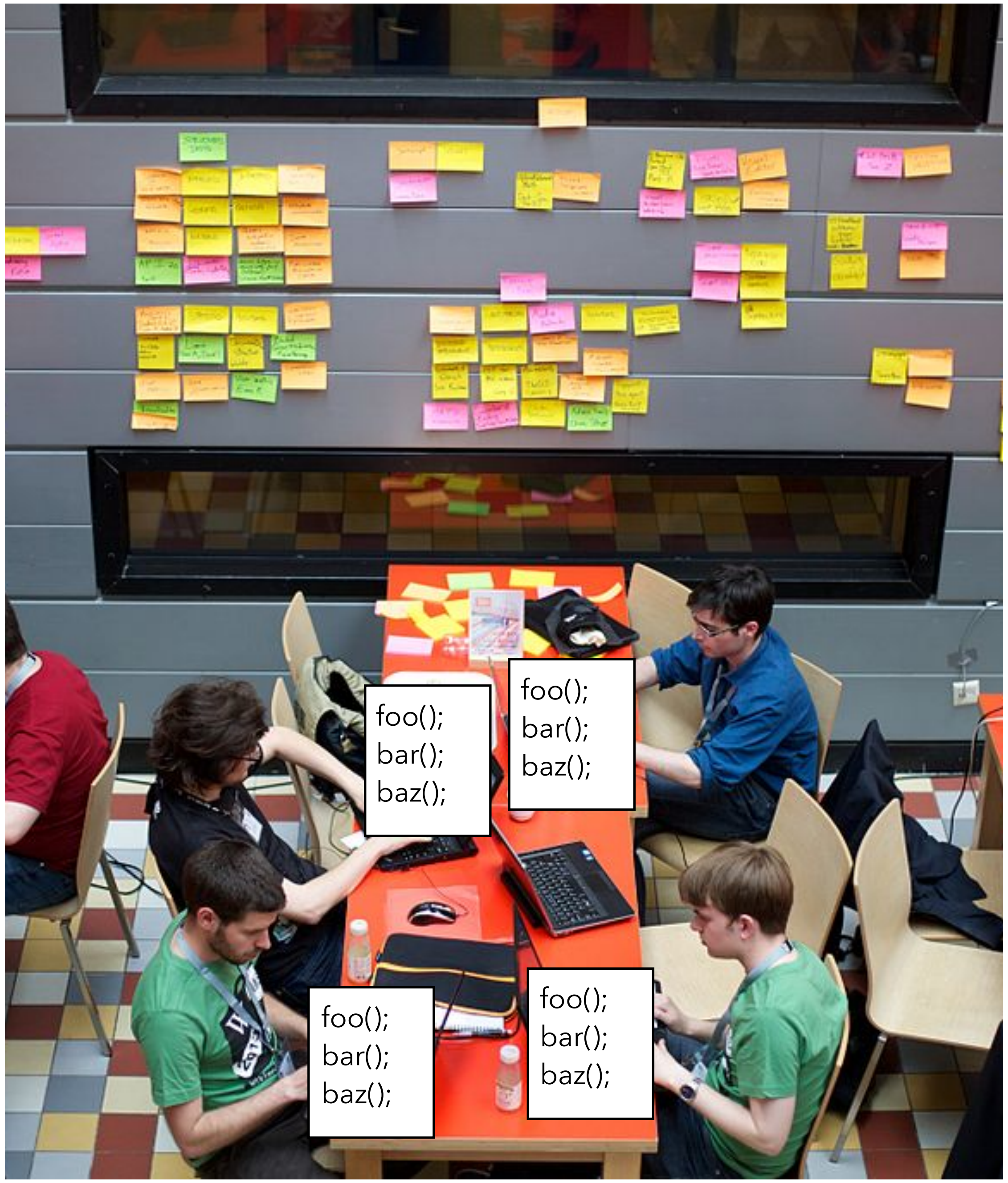


Mining



- Units of analysis
  - Code
  - Commits
  - Issues
  - Dependencies
  - Defects

Mining



- New science about process, method, architecture, domain, defects, debt
- Prediction techniques
- New analysis methods

Two sides of the same phenomenon

Comprehension

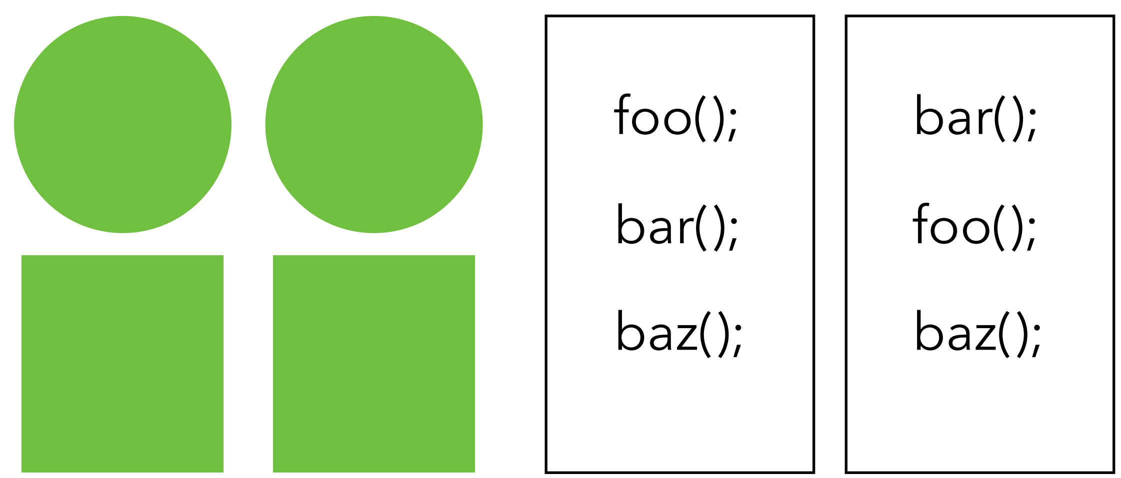
perception

cognition

decisions

collaboration

contexts



Mining

code

commits

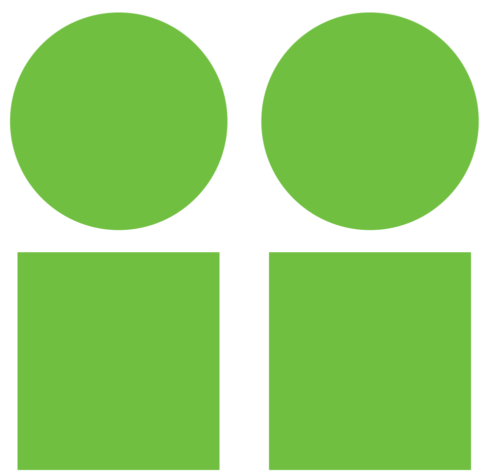
issues

dependencies

defects

Comprehension = better decisions

Comprehension
perception
cognition
decisions
collaboration
contexts



- Tools optimized to enhance comprehension
- Processes optimized to streamline collaboration
- Descriptive and predictive theories of comprehension that support design and education

Mining = better modeling

- Better predictions
- Better models of software process
- Better tools for software analytics

```
foo();
bar();
baz();
```

```
bar();
foo();
baz();
```

Mining
code
commits
issues
dependencies
defects

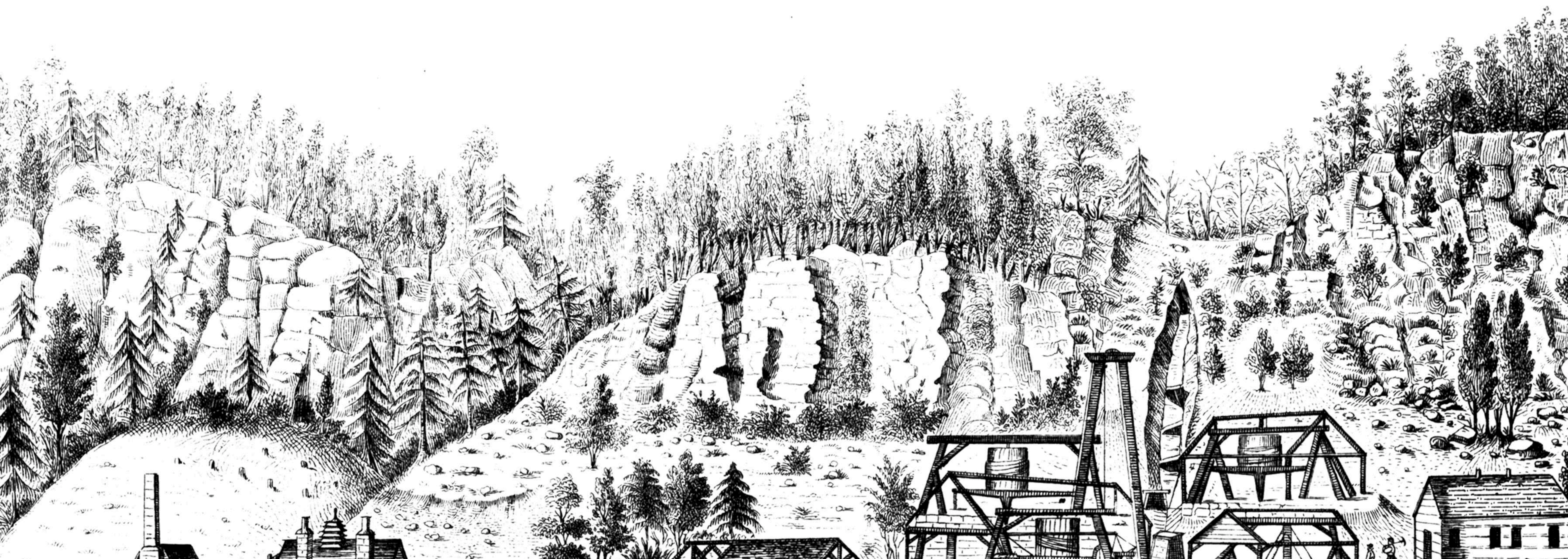
Disciplinarity is productive

- By focusing on *comprehension*, ICPC can enhance developers' understanding of complex systems
- By focusing on *mining*, MSR can can enhance developers' processes
- Neither of these necessarily require contributions from the other to be valuable

Four missed interdisciplinary opportunities

- Mining the mind
- Minding the mine
- Theory
- Grander challenges

Mining the mind



The problem

- Many ICPC studies are **small sample lab** studies
- Of 16 pre-prints this year, 6 include studies with human subjects
  - Recruited between 8 and 88 participants
  - All short tasks, interviews, or surveys
- Many of these studies need longitudinal, ecologically valid contexts to strongly support their claims

An ICPC example

- Tymchuk et al’s "JIT Feedback – What Experienced Developers like about Static Analysis.” ICPC ’18.
    - Solid interview study of 29 Smalltalk developers about a static analysis tool
    - Great for understanding developers’ **sentiments** about the tool
    - *Not* great for understanding **impact** of the tool, because it relied on retrospective self-report

A solution

- Measure comprehension *at scale* with repositories
- Repositories offer longitudinal, ecologically valid, ground truth contexts in which to test hypotheses
- In fact, ICPC is doing this already: 10 pre-prints actually used repositories—just not to understand program comprehension.

An approach

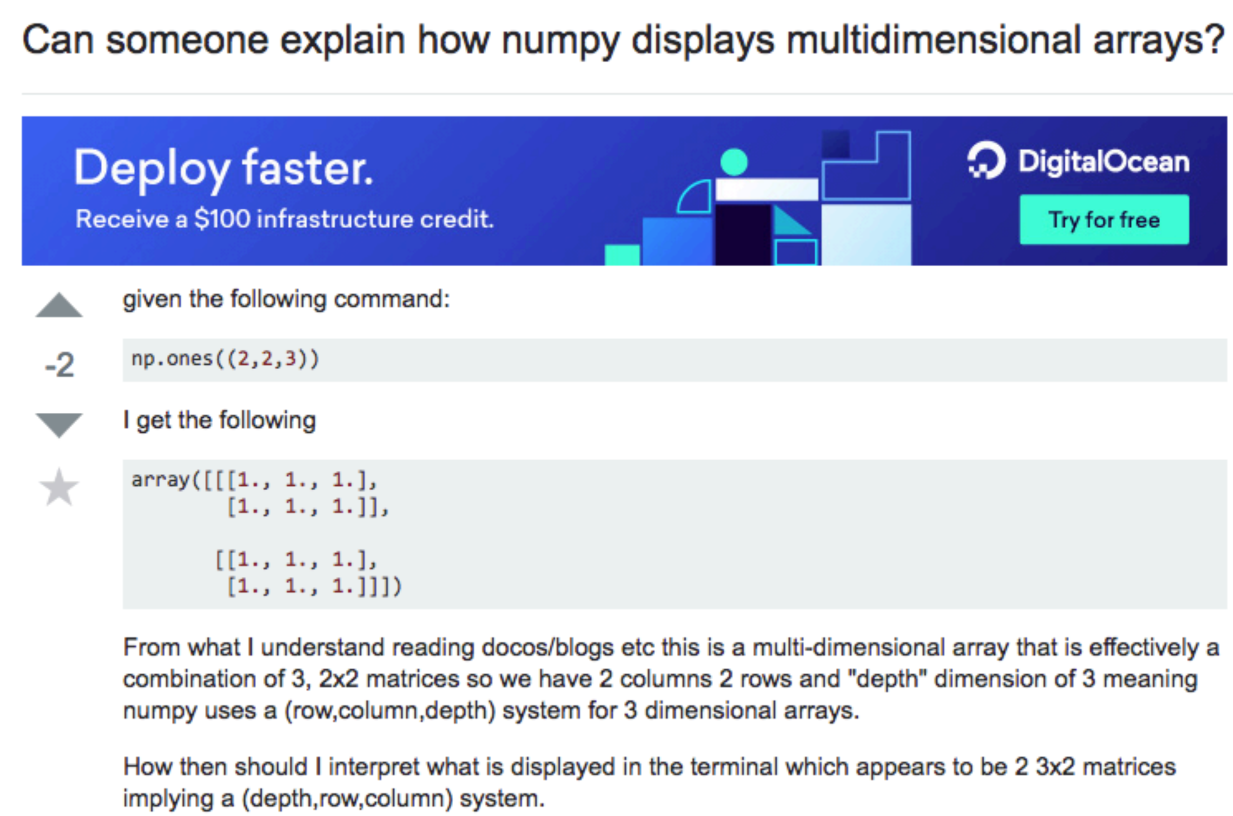
- Repositories hold traces of developers' *comprehension* of code
  - Defects may indicate **failure** to comprehend
  - Communication may indicate comprehension **needs**
  - Complexity may suggest comprehension **barriers**
- Few studies try to *model* these indicators of comprehension

Example: APIs & defects

- **Theory**
  - Hidden semantics result in developers with brittle comprehension of API semantics, who then write brittle code
  - e.g., many users of the Facebook React framework don't understand which calls are asynchronous, which leads to code that seems correct with shallow testing
- **Hypothesis**
  - More hidden the API semantics, more defects

Example: APIs & defects

- **Method**
  - Measure how hidden semantic facts are by counting the number of Stack Overflow questions about that API
  - Measure defect density of components
  - Correlate



Can someone explain how numpy displays multidimensional arrays?

-2

given the following command:

```
np.ones((2,2,3))
```

I get the following

```
array([[[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]]])
```

From what I understand reading docos/blogs etc this is a multi-dimensional array that is effectively a combination of 3, 2x2 matrices so we have 2 columns 2 rows and "depth" dimension of 3 meaning numpy uses a (row,column,depth) system for 3 dimensional arrays.

How then should I interpret what is displayed in the terminal which appears to be 2 3x2 matrices implying a (depth,row,column) system.

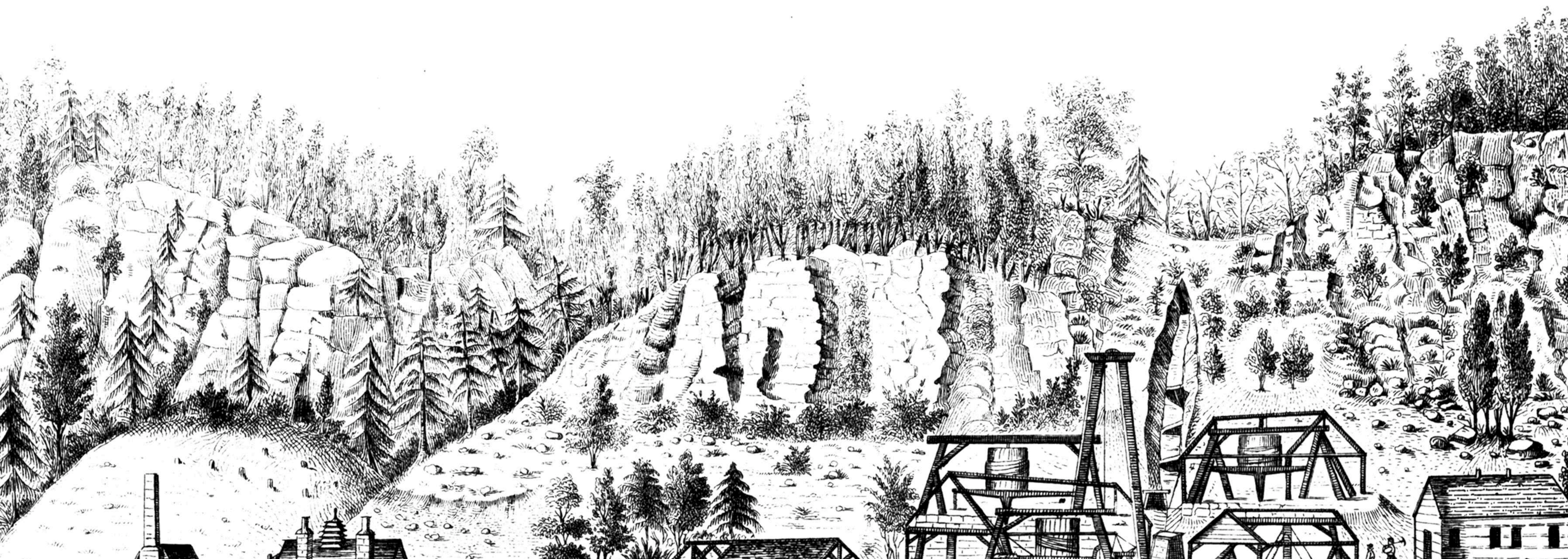
Example from MSR ‘18

- Some at MSR are already doing this!
  - Gopstein et al. “Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild.” MSR 2018
  - Operationalizes an indicator of comprehension
  - Shows a strong correlation between “confusing” patterns and bug-fix commits

Impact of mining the mind

- Longitudinal, community-wide measures of program comprehension
- Descriptive and predictive models of a community or organization’s comprehension gaps
- Associations between comprehension, defects, productivity, and other outcomes

“Minding” the mine



The problem

- Many MSR (and ICPC) papers do a great job testing **feasibility**, **correctness**, **coverage**, **accuracy** of tools
- However, of 11 pre-prints at MSR '18 that evaluated tools intended for developers, only one evaluated **usefulness**
- This bias towards **applicability** overlooks critical questions about how these tools would be *used* by developers, managers, and teams to actually improve software engineering.
- Leaves many fundamental premises about the utility of mining tools *untested*.

An MSR example

- Rath et al. "*Analyzing Requirements and Traceability Information to Improve Bug Localization*" MSR 2018.
  - Clever use of previously fixed bug reports to improve localization!
  - Robust evaluation against 13,000 bug reports
  - No evaluation of whether a ranked list of source files is **useful** to developers in comprehending, localizing, or repairing defects.

A solution

- We need to **test** these unverified premises with real developers on real teams
- Example premises to test:
  - *Managers want to analyze their team's activity*
  - *Predictions are trusted and actionable*
  - *Patterns in source code lead to valuable insights*
  - *Patterns in communication lead to valuable insights*
- When are these true? When are they not? Why?

An approach

- Putting tools in front of real developers, managers, and teams
- Show them our vision of how mining tools can be used to impact software engineering practice
- Elicit their questions, concerns, and ideas
- Better yet, *deploy* mining tools into practice, evaluating how they do and do not support software engineering

Example: prediction actionability

- **Theory**
  - Decision sciences shows that people generally don't use data to make decisions, they use it confirm prior beliefs
- **Hypothesis**
  - Developers and managers will view fault localization predictions as evidence of their prior knowledge about components, and see little actionable insight

Example: prediction actionability

- **Method**
  - Recruit 30 open source developers
  - Present fault localization source file rankings
  - Challenge developers to extract novel actionable insights from the data

Example: prediction actionability

- **Implications**
  - If my hypothesis is true, many mining tools that make predictions will be viewed as useless
  - May need to reconsider what output would be valuable to developers and managers
  - May need to invent new algorithms and tools to achieve usefulness

Example from ICPC '18

- Tymchuk et al's "JIT Feedback" paper we just discussed is a perfect example of a human subjects study of developers' **perception of value** of a tool's output
- Provides rich insights about precisely which rules were valuable, which rules were not, and why

Evaluating with human participants

- Many skills required to evaluate tools with people.
- My collaborators Thomas LaToza and Margaret Burnett and I have written down many of these skills for you.
    - Ko, A. J., Latoza, T. D., & Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. Empirical Software Engineering, 20(1), 110-141.


Empirical Software Engineering
February 2015, Volume 20, Issue 1, pp 110–141 | Cite as

A practical guide to controlled experiments of software engineering tools with human participants

Authors Authors and affiliations

Andrew J. Ko, Thomas D. LaToza, Margaret M. Burnett

Article
First Online: 27 September 2013

21 Shares 2.1k Downloads 20 Citations

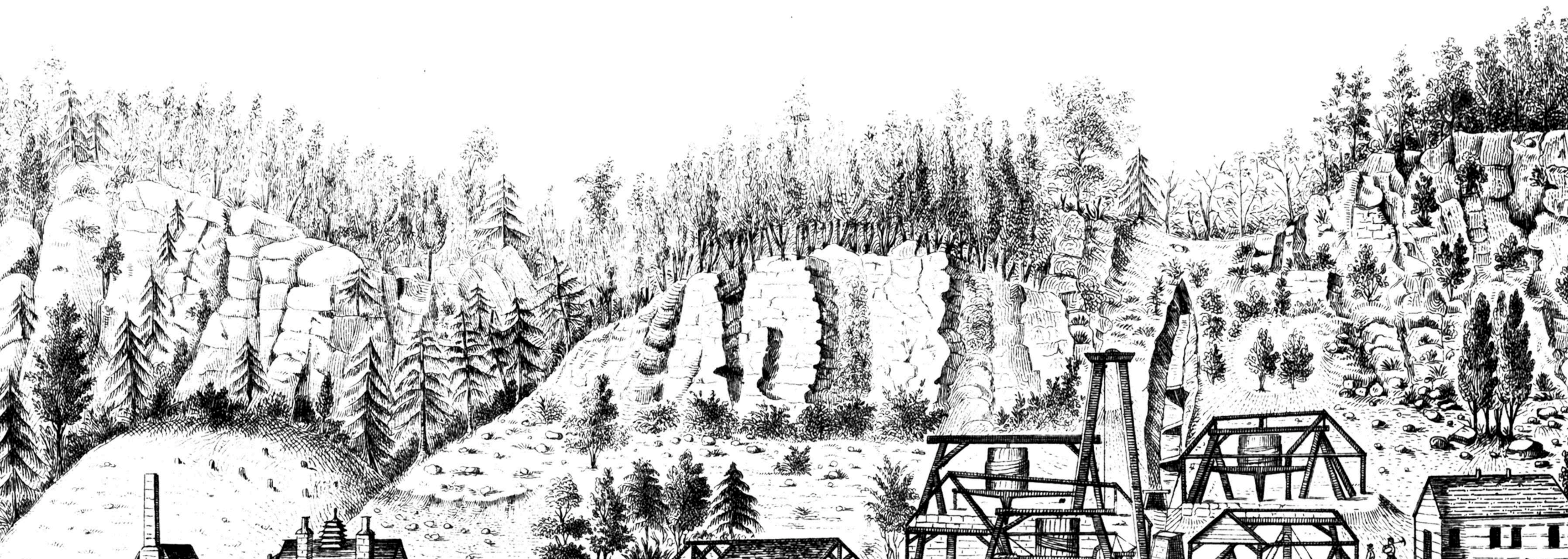
Abstract

Empirical studies, often in the form of controlled experiments, have been widely adopted i
software engineering research as a way to evaluate the merits of new software engineering
tools. However, controlled experiments involving *human participants* actually *using* new t
are still rare, and when they are conducted, some have serious validity concerns. Recent
research has also shown that many software engineering researchers view this form of tool
evaluation as too risky and too difficult to conduct, as they might ultimately lead to inconcl
or negative results. In this paper, we aim both to help researchers minimize the risks of thi
form of tool evaluation, and to increase their quality, by offering practical methodological
guidance on designing and running controlled experiments with developers. Our guidance
gaps in the empirical literature by explaining, from a practical perspective, options in the
recruitment and selection of human participants, informed consent, experimental procedu
demographic measurements, group assignment, training, the selecting and design of tasks,
measurement of common outcome variables such as success and time on task, and study
debriefing. Throughout, we situate this guidance in the results of a new systematic review
tool evaluations that were published in over 1,700 software engineering papers published f
2001 to 2011.

Impact of “minding” the mine

- Demonstrably useful software analytics tools
- A new science of software analytics decisions
- New tool requirements requiring further research
- More impact on practice

Theory



The problem

- Most ICPC studies **describe** or **predict** behaviors, practices, strategies, effects of tools; *few explain*.
- Most MSR studies **describe** or **predict** patterns, associations, and trends; *few explain*.
- None of the pre-prints in ICPC or MSR '18 had formal or informal theories that informed tool or empirical study design, or interpretations of results.
- Without **explanations**, all we have is a loosely connected set of empirical patterns, with no greater **theory** of how they relate
- We need theory to build upon each others' discoveries.

A solution

- We must produce **theories** that *explain* the major phenomena in software engineering (e.g., comprehension, process, coordination, defects)
- We must rigorously explain *why* defects occur, *why* builds fail, *why* decisions are poor, *why* projects are late, etc.
- By generating these explanations, we can derive hypotheses, and test them in the lab and the field, with developers and with data.
- Theories will then allow us to *combine* our results, and communicate greater truths to industry about software

An example theory from SE

- James Herbsleb's **Socio-Technical Theory of Coordination** (STTC) (Herbsleb 2016) .
- Explains how teams coordinate work, arguing that:
    1. Software is an interdependent network of *decision constraints* imposed by *technical dependencies*
    2. Teams, process, and modularity are all efforts to *align* coordination requirements determined by these constraints with actual coordination between individuals.

STTC in simpler terms

- If
  - developer A owns function foo(), and
  - developer B owns function bar(), and
  - foo() calls bar()
- Developers A and B must talk to each other about foo() and bar() to coordinate the dependency.

Support for STTC

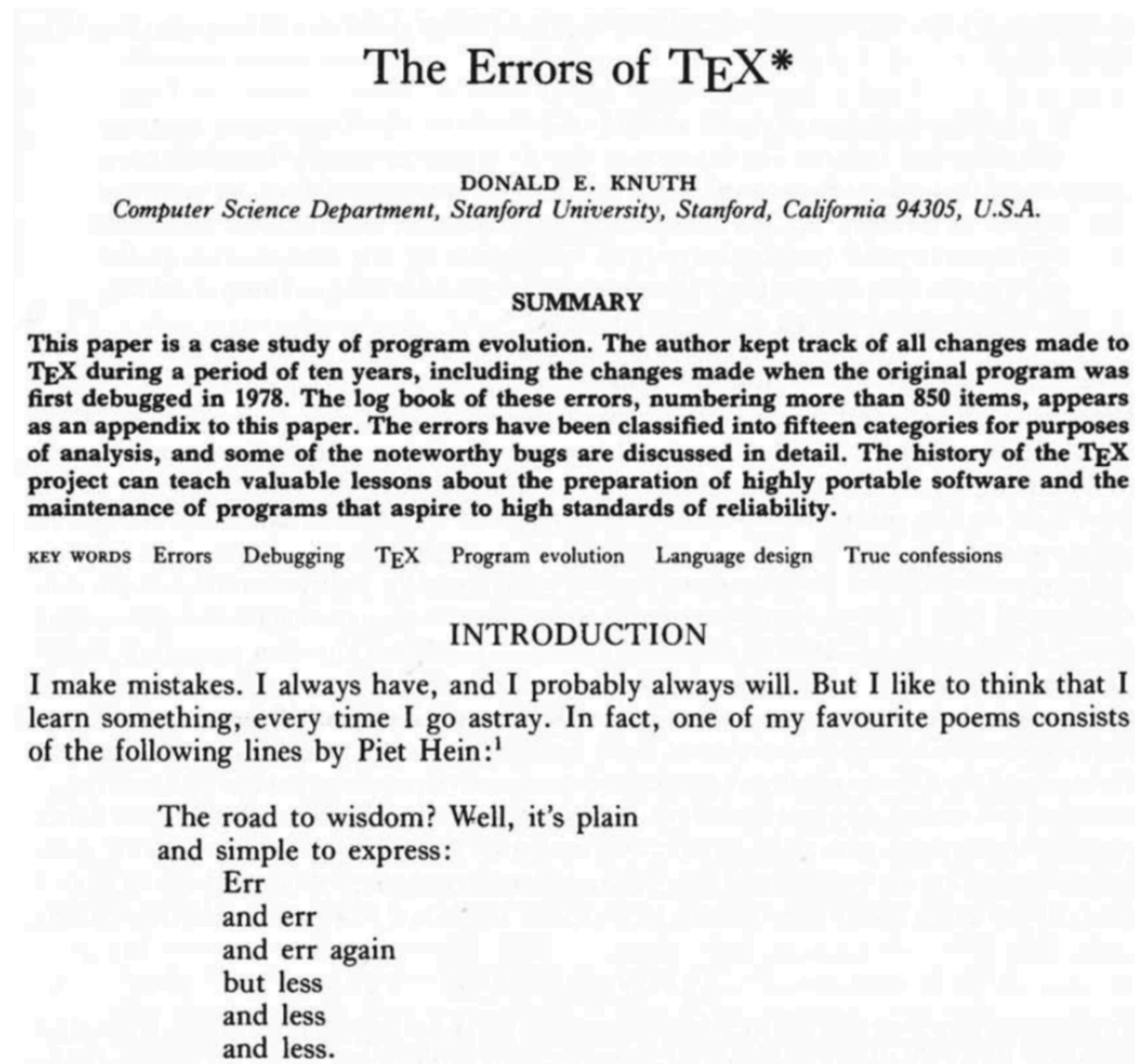
- The theory predicts that *misalignment* between social and technical constraints causes defects and delays by limiting the information that developers have for decision making.
- Evidence supports these predictions:
    - *Cataldo et al. 2008*: misalignment is related to time to resolve modification requests
    - *Cataldo and Herbleb 2012*: misalignment explained increases in software failures over time

Applying STTC

- Everyone in the room investigating questions of coordination should be attempting to falsify this theory:
    - Interpret prior work
    - Derive hypotheses
    - Test hypotheses
    - Interpret results
    - Connect results to prior work
- Allows us to integrate our individual publications into a greater whole, *explaining* the work of software engineering

A theory of defects

- Knuth's "Errors of TeX" (1989) is one of my favorite qualitative empirical studies from SE
- An epic-10 year diary study of defects
- Inside it is a fascinating theory of how defects arise in practice



The Errors of TEX*

DONALD E. KNUTH
Computer Science Department, Stanford University, Stanford, California 94305, U.S.A.

SUMMARY

This paper is a case study of program evolution. The author kept track of all changes made to TEX during a period of ten years, including the changes made when the original program was first debugged in 1978. The log book of these errors, numbering more than 850 items, appears as an appendix to this paper. The errors have been classified into fifteen categories for purposes of analysis, and some of the noteworthy bugs are discussed in detail. The history of the TEX project can teach valuable lessons about the preparation of highly portable software and the maintenance of programs that aspire to high standards of reliability.

KEY WORDS Errors Debugging TEX Program evolution Language design True confessions

INTRODUCTION

I make mistakes. I always have, and I probably always will. But I like to think that I learn something, every time I go astray. In fact, one of my favourite poems consists of the following lines by Piet Hein:[1]

The road to wisdom? Well, it's plain
and simple to express:
Err
and err
and err again
but less
and less
and less.

A theory of defects

- These actually map neatly on to more basic research on human error (Reason 1990), which I adapted into a theory of defects
    - Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages & Computing, 16(1-2), 41-84.

A framework and methodology for studying the causes of software errors in programming systems

Andrew J. Ko*, Brad A. Myers

Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA

Received 1 January 2004; received in revised form 1 July 2004; accepted 1 August 2004

Abstract

An essential aspect of programmers' work is the correctness of their code. This makes current HCI techniques ill-suited to analyze and design the programming systems that programmers use everyday, since these techniques focus more on problems with learnability and efficiency of use, and less on error-proneness. We propose a framework and methodology that focuses specifically on errors by supporting the description and identification of the causes of software errors in terms of chains of cognitive breakdowns. The framework is based on both old and new studies of programming, as well as general research on the mechanisms of human error. Our experiences using the framework and methodology to study the Alice programming system have directly inspired the design of several new programming tools and interfaces. This includes the Whyline debugging interface, which we have shown to reduce debugging time by a factor of 8 and help programmers get 40% further through their tasks. We discuss the framework's and methodology's implications for programming system design, software engineering, and the psychology of programming.

Explaining defects

- Argues that defects come from 5 sources
    1. Failure to attend closely to routine action (e.g., choosing an item in code completion)
    2. Misapplication of a rule in a novel context (e.g., using a for loop increment template for a decrement problem)
    3. Use of a bad rule (e.g., using for loops instead of iterators)
    4. Incomplete information about a problem space (e.g., brittle knowledge of an API's expressiveness)
    5. Problem space is too large to comprehend (e.g., reasoning about human behavior in a driverless car context)

Testing a theory of defects

- **Theory**
  - Failure to attend closely to a routine action causes defects.
- **Hypothesis**
  - Developers read and write a lot of routine *for()* loops. When those loops *deviate* from routine, developers will overlook this deviation, leading to defects.
- **Method**
  - Measure the defect density of functions and the deviancy of their *for()* loops, then correlate density to deviancy

Testing a theory of defects

- If we all spent time developing and testing this theory, we may produce a *grand theory* of where all defects come from
- Could use to reliably predict when defects will occur, helping to prevent them through training, process, and tools

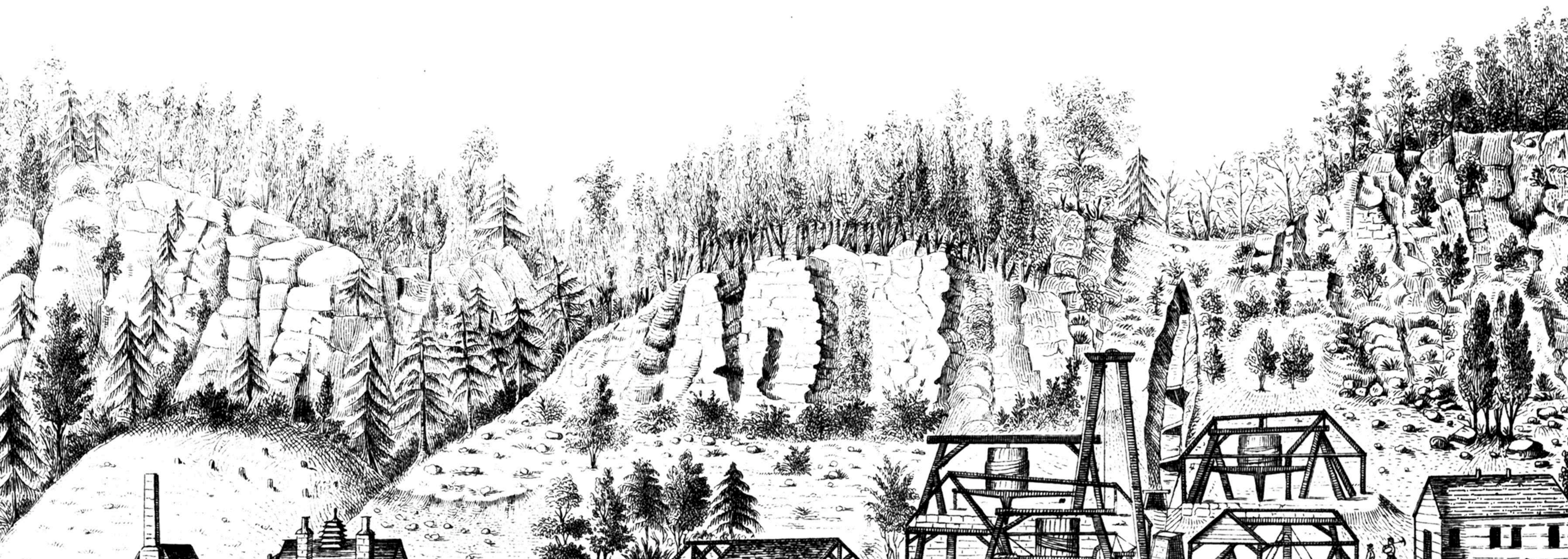
Theory for tools

- Theory isn't just for empirical studies
- Tools *embody* theories of programming
  - e.g., the implicit theory of defect prediction tools is that developers and teams need help localizing defects and prioritizing testing
  - Is this theory true?

A theoretical call to action

- *All* research on comprehension and mining, empirical or technical should advance or falsify a *theory* about software engineering
- If we all do this, then we have a common framework in which to combine our individual discoveries into greater truths

Grander challenges



The problem

- Developers don't see value in much of our research
  (Lo, Nagappan, Zimmermann 2015)
- According to 512 practitioners at Microsoft, 29% of our research ideas are not *not actionable*, *not useful*, *not generalizable*, or *too costly*
- No correlation between what developers' valued and what we cite in research papers

A solution

Ko, A. J. (2017, May). A three-year participant observation of software startup software evolution. ICSE, SEIP.

- Focus on the big questions that industry can't answer
- Here are some questions CTO's wanted research to answer
  - *How can I know a new software process will help?*
  - *How can we onboard new developers faster?*
  - *How can my developers learn APIs faster?*
  - *How can I align my technical decisions with business priorities?*
  - *How can I know what's happening in the field if no one reports it?*
  - *How can I discover single points of failure?*

We can answer these, but we need both comprehension *and* mining

- *How can I know a new software process will help?*
- *How can we onboard new developers faster?*
- *How can my developers learn APIs faster?*
- *How can I align my technical decisions with business priorities?*
- *How can I know what's happening in the field if no one reports it?*
- *How can I discover our single points of failure?*

- But also organizational scientists, management scientists, and learning scientists
- We should be bringing together interdisciplinary teams to answer these big questions

Example: onboarding

- Millions of developers start new jobs every year, but aren't productive for months.
- How can we help them onboard faster?
  - We have a few studies of onboarding (e.g., Begel & Simon 2008) that suggest organizational management theories of "newcomer socialization" best explain learning needs
  - New developers need mentors, models for proper behavior, connections to expertise about architecture, code review practices, norms about meetings, walkthroughs of feature implementations, and much more

Example: onboarding

- One idea from this study was *feature interviews*, in which a new hire meets with a developer to learn about:
  - The features the developer owns
  - The architecture of the features
  - How the features are situated in the larger architecture
- These could be supported by a new class of **architectural walkthrough tools** that situate features in architectures, provide rationale, reveal business goals, and surface practices and norms around process

Example: onboarding

- How can **comprehension** help? Answer these:
  - How can developers *author* a walkthrough to reveal this information in a feature interview?
  - How can we know if an authored walkthrough will produce effective comprehension of architecture?
  - What data other than code will be necessary to surface in such a walkthrough?
- These are *foundational* program comprehension questions that go well beyond reading code.

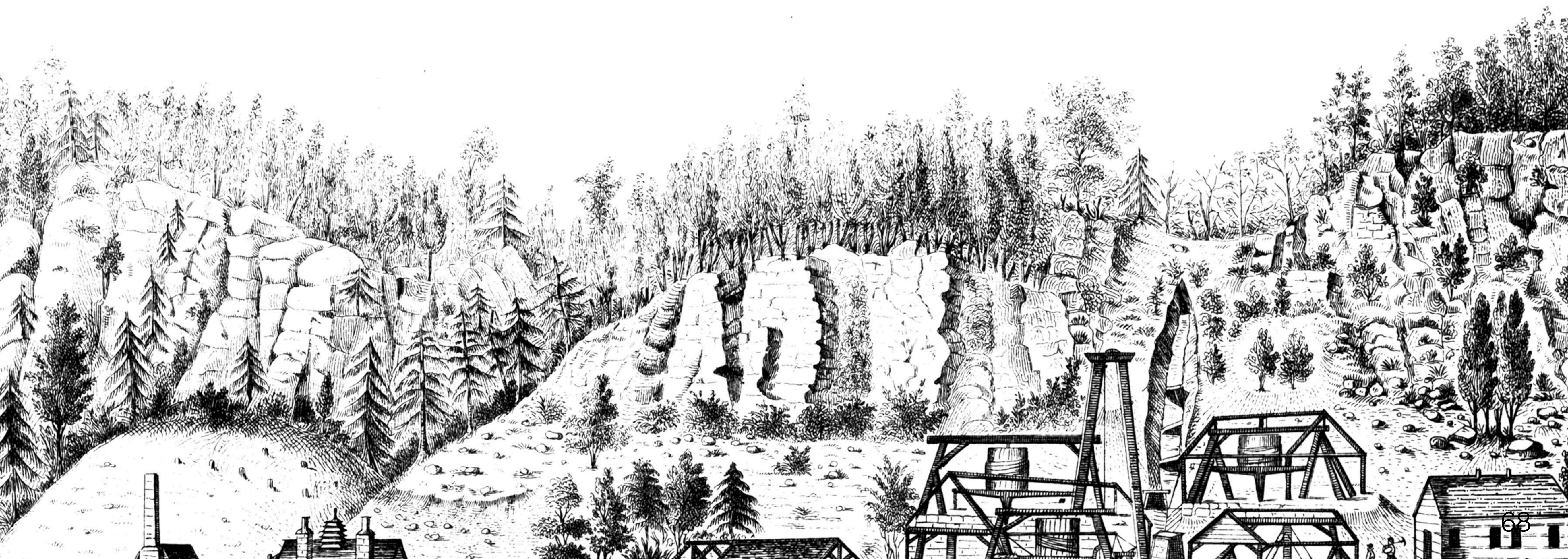
Example: onboarding

- How can **mining** help? Answer these:
  - What kinds of project history are necessary for comprehending architectural rationale?
  - Can we help a developer preparing for a walkthrough predict what code is necessary to discuss?
  - How can we use contribution history to recommend who is qualified to author a feature walkthrough?
- These are foundational questions about prediction and mining that go well beyond repositories.

This is atypical SE research

- Requires us to tackle phenomena we don't usually study (organizations, learning, teaching, business decisions)
- Tackling the real struggles that industry has requires **interdisciplinary expertise**
- Research contributions may not look like the technical and empirical contributions we typically value in software engineering research
- It might instead advance theories of organizational learning, designs in HCI, strategies in computing education

Next steps



Make time

- To aim this high, we have to think about more than the next paper or promotion
- Some of these problems might take *multiple years* before we have progress worth reporting
- If you have tenure, use it to think bigger, broader, and longer

Be inclusive

- Technical contributions matter
- But to make progress on these big problems, we *must* value other forms of scholarship (theory, development of instruments, etc.)

Read other disciplines

HCI, organizational science, management science, cognitive psychology, social psychology, and others are **explaining** the software engineering phenomena that our field is investigating. We should know what they've discovered, and build upon it.

Connect with software engineers and CTOs

Visit local meetups. Talk to them about what's hard about their jobs. Discover what questions they have. You'll be surprised how little their needs align with our questions.

Connect MSR and ICPC

You have more in common than you think.
Use this week to find a shared project.

The cost of inaction

- If we don't pursue interdisciplinary work, our field may become irrelevant
- We **must** show the world that the questions we answer in software engineering matter not only to CS, but software engineering practice
- We must also show relevance to other fields struggling with software development:
    - Medicine, natural sciences, public policy, law, etc. all need our help, but we put most of our attention on a few specific safety-critical domains

If we do this, our work will be deeper and more impactful

Thanks!

Andy J. Ko, Ph.D.

Summary

- ICPC and MSR study the same thing with different lenses.
- The mining lens can increase comprehension's scale, rigor
- The comprehension lens can increase mining's relevance
- Both mining and comprehension need theory for progress
- Both need to ask bigger, more relevant questions
- This requires us to do interdisciplinary work and reach outside of *academia*