Making Hay from Wheats: A Classsourcing Method to Identify Misconceptions

Siddhartha Prasad
Computer Science Department
Brown University
Providence, Rhode Island, USA
siddhartha_prasad@brown.edu

Ben Greenman
Computer Science Department
Brown University
Providence, Rhode Island, USA
benjamin.l.greenman@gmail.com

Tim Nelson
Computer Science Department
Brown University
Providence, Rhode Island, USA
timothy_nelson@brown.edu

John Wrenn
Computer Science Department
Brown University
Providence, Rhode Island, USA
jack@wrenn.fyi

Shriram Krishnamurthi
Computer Science Department
Brown University
Providence, Rhode Island, USA
shriram@brown.edu

ABSTRACT

Novice programmers often begin coding with a poor understanding of the task at hand and end up solving the wrong problem. A promising way to put novices on the right track is to have them write examples first, before coding, and provide them with feedback by evaluating the examples on a suite of *chaff* implementations that are flawed in subtle ways. This feedback, however, is only as good as the chaffs themselves. Instructors must anticipate misconceptions and avoid expert blind spots to make a useful suite of chaffs.

This paper conjectures that novices' *incorrect examples* are a rich source of insight and presents a classsourcing method for identifying misconceptions. First off, we identify incorrect examples using known, correct *wheat* implementations. The method is to manually cluster incorrect examples by semantic similarity, summarize each cluster with a potential misconception, and use the analysis to generate chaffs—thereby deriving a useful by-product (*hay*) from examples that fail the wheats. Classsourced misconceptions have revealed expert blind spots and drawn attention to chaffs that seldom arose in practice, one of which had an undiscovered bug.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

problem understanding, examples-first, Examplar

ACM Reference Format:
Siddhartha Prasad, Ben Greenman, Tim Nelson, John Wrenn, and Shriram Krishnamurthi. 2022. Making Hay from Wheats: A Classsourcing Method to Identify Misconceptions. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research (Koli 2022), November 17–20, 2022, Koli, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3564721.3564726

1 EXECUTABLE EXAMPLES, WHEATS, AND CHAFFS

Before novices begin coding the solution to a programming problem, it is important that they understand the problem to avoid losing time and missing learning objectives. Unfortunately, misconceptions can hinder understanding [14, 16, 21].

A promising way to detect misconceptions and quickly provide feedback is via executable examples, as implemented by tools such as CodeWrite [6] and Examplar [22, 23]. In Examplar, the method is to ask novices for examples and to provide feedback on their example suites using hidden correct (*wheat*) and incorrect (*chaff*) solutions. Students are given feedback on *validity* (did all examples pass the wheat?) and *thoroughness* (how many chaffs did they catch?).

Consider the task of computing the median of a list of numbers. Examples consist of input lists and output numbers. The first column in table 1 presents three examples. The remaining columns are based on one wheat, which correctly implements median, and two chaffs, which return the average (mean) and middle element (middle) of a list. These chaffs are a good source of feedback because they agree with the wheat on some examples, but not all. The challenge for students is thus to write a set of examples that catches each of the chaffs (by causing them to fail) without failing the wheat. Collectively, the examples in table 1 meet these requirements. Examplar would give this example suite positive feedback about problem understanding: in particular, the student has not mistaken median for mean or assumed that the input is sorted (as many math texts implicitly do!). In contrast, if a student writes *only* the first example, they may have one of these two problem misunderstandings (the second of which, in particular, we have seen in practice!).

Of course, the quality of feedback provided by wheat and chaff solutions depends critically on the wheats and chaffs themselves. Chaffs, in particular, are hard to design. On one hand, the set of chaffs should cover standard student misconceptions. If table 1 had omitted the chaff for mean, then students who confused median

Koli 2022, November 17–20, 2022, Koli, Finland

ACM ISBN 978-1-4503-9616-5/22/11…$15.00
https://doi.org/10.1145/3564721.3564726

Table 1: Executable examples, a wheat, and two chaffs for a median function

Example	Wheat: median	Chaff 1: mean	Chaff 2: middle
`median([1, 2, 3]) is 2`	✓	✓	✓
`median([1, 2, 6]) is 2`	✓	× (mean is 3)	✓
`median([2, 1, 3]) is 2`	✓	✓	× (middle is 1)

This is a good set of examples because:

- every example agrees (✓) with the wheat, and
- every chaff has at least one example that it disagrees with (×).

with mean would get no feedback to correct their understanding, and might implement a completely different (and quite wrong) function. On the other hand, a large set of chaffs can frustrate students. Students have a tendency to want to catch all chaffs, so the more there are, the longer they spend [22]. This distracts from function-writing, leads to diminishing returns, and worse, may not even uncover real misconceptions. Thus, good chaff construction is critical.

Unfortunately, there are no concrete tools to help instructors design chaffs. There are several rules of thumb suggested by the lead designer of Examplar, such as *avoid difficult-to-catch chaffs* and *favor logical errors over programming errors* [24]. However, this leaves open how to find salient errors for a new problem. In particular, when chaffs are designed by course staff, they can suffer from expert blind spots.

In this paper, we propose a lightweight method for identifying potential misconceptions that builds on wheats and chaffs by analyzing an underused resource: examples that fail the wheats. Prior work by the Examplar team ignored these examples because there are several *uninteresting* reasons for an example to fail a wheat (personal communication). The example might have a typo in the input/output data; it might contain a basic logical error; and worst of all it might be an attempt to game the system with a nonsense example that trivially catches all the chaffs. However, if there are good-faith failures, then perhaps they can be mined to reveal patterns of mistakes for chaff construction.

This paper's contributions are as follows:

- We find that numerous wheat failures (about 30%) appear to be good-faith efforts based on misunderstandings.
- We propose a method for identifying potential misconceptions from input/output examples (section 2): start with wheat failures, group them using (manual) semantic clustering, and use the clusters to guide the design of chaffs.
- We validate the method by comparing our classsourced chaffs to expert-designed chaffs written by course staff (section 4). Classsourcing reveals expert blind spots and provides support, or a lack thereof, for expert chaffs.
- We present evidence that classsourcing is useful for more than programming problems by applying the method to a data structure specification problem in formal logic (section 5). Furthermore, we present a talk-aloud in which peer students applied labels to incorrect data structure examples for clustering. The results of this study suggest that students themselves can effectively contribute to the time-intensive labelling process.

2 CLASSSOURCING FOR CHAFFS

Our proposal is to analyze students' examples for evidence of misconceptions. Examples that fail the wheats are a particularly rich source of insight because they are often based on a misunderstanding. For instance, the following two examples for `median` fail the wheat:

`median([1, 2, 6]) is 3`   and   `median([9, 3, 3]) is 5`

They agree with the behavior of `mean`, however, which suggests that the author of these examples has misunderstood the definition of the median problem.

The key steps for classsourcing are logging, clustering, and generalizing:

1. *Log* example suites whenever students request feedback. Pay close attention to examples that fail the wheats. (Examples that fail *some but not all* wheats test unspecified behavior [23].)
2. Separate typo-level errors and adversarial examples from non-trivial, good-faith errors. *Cluster* the latter into semantically similar groups.
3. *Generalize* each cluster into a chaff. The chaff should agree with every example in the cluster without overfitting.

Clustering and generalizing require significant insight. Given the two examples for `median` above, an expert would need to group them into a cluster (along with other logged examples) and generalize the cluster with an implementation of the `mean` function. However, the overall burden on experts is much lower than in the state of the art [6, 22], which is entirely expert-driven, offering no guidance on what the misconceptions could even be.

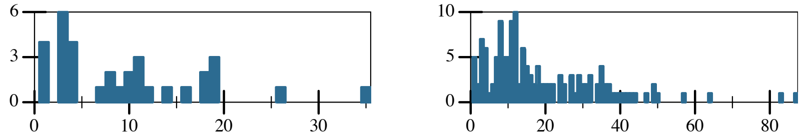
3 STUDY CONTEXT, DATASETS, AND ANALYSIS METHOD

The primary validation for our classsourcing method is from two programming problems. Both problems were deployed in Fall 2020 at a highly selective, private US university as part of an accelerated introductory computer science course.

- *Nile* is a collaborative filtering problem that takes place in the context of a hypothetical online bookstore. Given a collection of recommended books, there are two tasks: compute top recommendations for a single book, and compute popular pairs of books across the entire collection.

Problem	# Wheat Failures	Coder 1	Coder 2	Cohen κ
Nile	319	149 (47%)	46 (31%)	0.85
DocDiff	1303	88 (7%)	26 (30%)	0.83

(a) Number of wheat failures, number of coded failures, and Cohen kappa agreement score



(b) Nile distribution of wheat failures

(c) DocDiff distribution of wheat failures

A bar at (x, y) means y students each submitted x wheat failures.

Figure 1: Wheat failures dataset

- *DocDiff* is a document-similarity problem. The task is to compute the overlap between two non-empty lists of strings by reducing them to bags of words and comparing the vectors [18].

For these assignments, students were graded on both their implementation and a final test suite. Students were encouraged, but not required, to start out by writing examples and using Examplar to check their understanding. These early examples could be submitted as part of the final test suite, but were not otherwise graded. Examplar was available as part of the standard Web-based IDE for the course. This IDE logged every feedback request to a database.

Dataset and Analysis Method. Figure 1a summarizes our datasets and analysis. Across the two assignments, we collected 1,622 input/output examples that failed the wheats: 319 from Nile and 1,303 from DocDiff. Two coders analyzed a sample of this data to determine why each example failed. Using techniques from grounded theory [8], they generated a labelling rubric for each assignment. Some labels corresponded to assignment-specific misconceptions, while others were generic, such as "typo" and "type error." For both problems, the coders achieved a Cohen κ over 0.8 (fig. 1a).

Due to the large number of wheat failures, the coders reviewed a sample of the full dataset. Sampling for Nile was done by first selecting a random student and then selecting a random (uncoded) example from the student. Sampling for DocDiff was similar, but because of the many students with a huge number of wheat failures (fig. 1c) all students whose total failures fell outside one standard deviation of the median were excluded. The Coder 1 column of fig. 1a shows how many entries were in each sample and, in parentheses, the percent of the full dataset that this number represents. After the first coder analyzed these samples, the second coder analyzed a subset of the coded entries drawn uniformly at random. The Coder 2 column shows the number of entries in each subset. For Nile, Coder 1 analyzed 47% of the wheat failures, Coder 2 analyzed 31% of these coded failures, and 53% of the failures were not reviewed by either coder.

Observe that sampling from the dataset is a reasonable method, if not optimal. Our goal is to find misconceptions that students have. If even a subset of the data reveals a potential misconception, that is an improvement. Adding more data cannot invalidate this newfound misconception, only reduce its relative frequency.

4 CLASSSOURCED CHAFFS VS. EXPERT-DESIGNED CHAFFS

Following the process in section 2, the two coders manually clustered wheat failures to identify misconceptions. None of the wheat failures seemed adversarial, but many were based on typos, type errors, or hard-to-categorize mistakes (27% in Nile and 54% in DocDiff). The rest of the wheat failures formed meaningful clusters with 1 to 13 failures each (mean: 3). The payoff of this analysis is a lower bound on the number of latent issues:

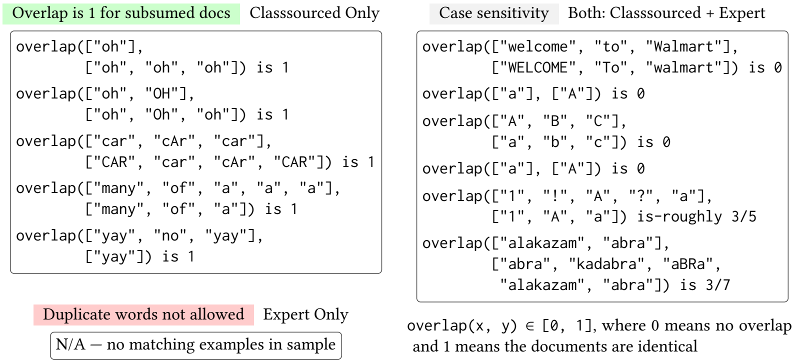
expert chaffs did not anticipate 25 issues that appeared in the classsourced data.

The expert chaffs also predicted a number of issues that never arose in our sample. Figure 2 summarizes the evidence for these conclusions. Classsourcing identified 17 potential misconceptions in Nile and 11 in DocDiff. The experts (course staff) missed most of these. In addition, the expert chaffs predicted several misconceptions that are not supported by the wheat failures: 6 in Nile and 3 in DocDiff. These chaffs might be supported by wheat failures we did not sample, but they may also be unnecessary.

	Nile	DocDiff
Classsourced Only	16	9
Both	1	2
Expert Only	6	3

Figure 2: Classsourced Chaffs (based on actual errors) vs. Expert Chaffs (based on predicted errors)

To illustrate, consider DocDiff. An input/output test for this problem consists of two input documents and a probability representing their overlap. Figure 3 presents three errors/chaffs related to this problem and the clusters of wheat failures (if any) that our two coders developed. One error revealed by classsourcing was that a large document and a fragment of the same should be regarded identical (overlap = 1). This incorrect belief was not anticipated by expert chaffs. In the other direction, one expert chaff rejected input documents with duplicate words; yet, no students made this error. Finally, an error that was both predicted by the experts and present in the data was that the similarity metric should be case-sensitive. Note that although the case-sensitivity cluster has six examples,



Overlap is 1 for subsumed docs — Classsourced Only

```
overlap(["oh"],
        ["oh", "oh", "oh"]) is 1
overlap(["oh", "OH"],
        ["oh", "Oh", "oh"]) is 1
overlap(["car", "cAr", "car"],
        ["CAR", "car", "cAr", "CAR"]) is 1
overlap(["many", "of", "a", "a", "a"],
        ["many", "of", "a"]) is 1
overlap(["yay", "no", "yay"],
        ["yay"]) is 1
```

Case sensitivity — Both: Classsourced + Expert

```
overlap(["welcome", "to", "Walmart"],
        ["WELCOME", "To", "walmart"]) is 0
overlap(["a"], ["A"]) is 0
overlap(["A", "B", "C"],
        ["a", "b", "c"]) is 0
overlap(["a"], ["A"]) is 0
overlap(["1", "!", "A", "?", "a"],
        ["1", "A", "a"]) is-roughly 3/5
overlap(["alakazam", "abra"],
        ["abra", "kadabra", "aBRa",
         "alakazam", "abra"]) is 3/7
```

Duplicate words not allowed — Expert Only

N/A — no matching examples in sample

`overlap(x, y) ∈ [0, 1]`, where 0 means no overlap and 1 means the documents are identical

Figure 3: Three preliminary misconceptions for DocDiff and the matching wheat failures

one of them is duplicated. This is because we happened to sample two similar wheat failures from the same student.

Overall, the agreement between classsourced chaffs and expert chaffs is remarkably small. Given that the classsourced chaffs are supported by actual errors rather than errors anticipated by the course staff (who had even refined their chaffs over several years), classsourcing fills an important gap between actual learners and experts' models of learners.

5 FOLLOWUP: CLASSSOURCING FOR SPECIFICATION PROBLEMS

As further validation of our classsourcing method, we applied it in a second domain: formal methods. The data for this study originated in a Spring 2021 course on applied logic for programmers that was offered at the same university (section 3). In this setting, an example is an instance or non-instance of a mathematical structure. Wheats and chaffs are mathematical specifications of structures. Students wrote examples as relations using a pedagogic variant of the Alloy modeling language [11, 12] called Forge [3, 19]. The specific problem we studied was the following:

- *Undirected tree*, or *U-Tree*, is a data structure specification problem. Given an English description of an undirected tree, the task is to formalize the specification in Forge.

For this assignment, students were required to submit both a final specification and a suite of example instances. The examples were graded on their performance against instructors' wheat and chaff specifications. Before the deadline, students could get feedback from either a command-line tool (similar to Examplar) or the course submission server. Unfortunately, we do not have access to the Examplar-level errors that students made. Instead, we can see *only* the wheat failures students uploaded to the submission server.

We found a total of 12 wheat failures among the submissions. In principle, there should be *no* such wheat failures, because students would have caught them earlier with the command-line tool! However, there were a few incentives to use the server for early attempts: (1) The command-line tool did not support machines with Apple M1 chips, forcing some students to use the server for feedback. (2) The server gave more feedback than the command-line tool by running a small test suite in addition to wheats and chaffs. (3) Submitting gave students a way to save their work; in case of emergency, the latest submitted attempt would receive at least partial credit.

5.1 Analysis

Two coders analyzed the 12 U-Tree wheat failures. Due to the small number of examples, each coder analyzed every failure. The coders achieved near-perfect agreement: they disagreed only on the specific label for one example that used an undeclared variable ("typo" vs "other"), which is an uninteresting disagreement.

Classsourcing revealed one unanticipated issue: that edges in an undirected tree need not be symmetric. This error was common because students wrote example trees as Forge relations and thus had to model each undirected edge with two directed edges. We were therefore surprised that this issue was not anticipated. Curiously: it was! However, there was a bug in the relevant chaff (fig. 4), which led to this problem not being caught. Thus, our process was able to find a bug in a chaff, which was an unexpected benefit.

Classsourcing found no evidence for three anticipated chaffs, one of which is the buggy one:

(1) *Connected asymmetry.* As mentioned above, one chaff meant to allow asymmetric trees but included a minimal-connectivity constraint that rejected basic asymmetric examples (fig. 4).
(2) *Self-edges.* None of the wheat failures used an edge to connects a node to itself.
(3) *Double edges.* None of the wheat failures used two paths to connect the same pair of nodes.

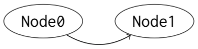
However, the very small number of wheat failures in the submission server does not by itself provide evidence that these chaffs are worth revising or removing for the next offering of this problem. We would need the logs from the command-line tool for that.

5.2 Can Non-Experts Apply Labels?

The most labor-intensive step in our process is labeling the wheat failures. Some of the work might be offloaded to non-experts, however, if experts can supply a preliminary set of labels and a rubric to guide their application.[1]

[1] How to compensate students for this work is left outside the scope of this paper.

```
example e is notSymmetric:
  Node = Node0 + Node1
  edges = Node0 -> Node1
```



(a) Asymmetric example; this is not a valid U-Tree

```
pred notSymmetric:      -- Goal: asymmetry ok
  Node->Node in *edges  --  Connected
  no iden & edges       --  No self-loops
  all n : Node |        --  Minimally connected
    all m : Node |
      (n->m + m->n) in edges
      implies
      (n->m + m->n) not in ^(edges - (n->m + m->n))
```

(b) Chaff that unintentionally rejects the asymmetric tree

Figure 4: Classsourcing revealed a buggy U-Tree chaff

To test the feasibility of this idea, we conducted a talk-aloud study of twelve students using the U-Tree data to seek out disagreements between students' labels and our own expert-applied labels. Participants were U.S. citizens or permanent residents who had taken the 2022 iteration of the logic course described above. They were compensated $50 for participating in a 25-minute study via Zoom. We were able to find 12 subjects;[2] the fact that we had 12 subjects and 12 wheat failures is a coincidence. The 2022 iteration did *not* assign the U-Tree problem; however, participants who had previously enrolled in the 2020 course may have seen the problem.

The study had two parts: training and testing. First, participants read an English specification of undirected trees and were asked to decide whether five proposed examples were valid or not. Participants received immediate feedback from the study software that explained the correct choice. Second, participants were given a set of labels and asked to categorize our 12 U-Tree wheat failures.

We recorded the training responses, but did not analyze them and did not exclude any participants based on their performance on the training questions. As for the testing task, most of the wheat failures (7/12) gathered a strong consensus among student labels that furthermore agreed with our label. For the other examples (5/12), student labels were divided. In four of these, the most popular student label agreed with the expert label. In the fifth one, the results are skewed by an issue among students: over 25% misunderstood the `set` multiplicity of Forge. (The talk-aloud interviews provided crucial evidence for this misunderstanding.) This preliminary work suggests that at least where there is consensus, student coders who have completed a related course can provide useful initial labeling.

6 THREATS TO VALIDITY

Internal Validity. Although our analysis suggests that some expert chaffs may be unnecessary (section 4), they could very well correspond to real misconceptions that are present elsewhere: in the unsampled part of the data; in student groups with different backgrounds; in similar students but under different instruction or even problem description; and so on. However, we believe all these reasons only strengthen the need for the approach described in this paper. A second threat is that our sampling did not account for duplicate wheat failures by the same student (mentioned in section 4), which means that the size of a cluster may not match the number of students who had the misconception. Our two-coder process is subject to internal threats to validity: e.g., the high agreement may be due to coders aligning their biases rather than eliminating them. Finally, the talk-aloud interviews (section 5.2) suggest that some participants misunderstood aspects of Forge and the U-Tree problem, which casts doubt on their choice of labels. It furthermore raises questions about how to train participants and when to ignore a response, and suggests that more work is needed in the design of languages like Forge and Alloy.

External Validity. Our datasets are relatively small. To generate truly robust chaffs, we would have to repeat this study under varying conditions. For U-Tree in particular, we have only a tiny dataset of 12 wheat failures. Nevertheless, the fact that this tiny dataset led us to discover a lurking error that was not found by other means—something the process was not even intended to help with—shows further value to this approach.

7 RELATED WORK

Problem understanding is a common issue among novice programmers. Both Whalley and Kasto [21] and Loksa and Ko [14] observe that novices often began coding with an incorrect plan and seldom thought to critique their plan relative to the problem statement. Prather et al. [16] report similar issues among programmers using an automated assessment tool (AAT), and suggest that the "single greatest weakness" (§5) of contemporary AATs is their lack of support for problem understanding.

Our method for discovering misconceptions is based on an examples-first style of programming, as supported by Examplar [22] and CodeWrite [6, 16], rather than on evaluating test suites (e.g., [2, 7, 15]; refer to [22] for a detailed comparison to testing). Examplar gives feedback on example suites using wheat and chaff solutions. CodeWrite asks students to predict the output of specific test cases. Though we have applied our method only in the context of Examplar, the analysis of incorrect predictions in Denny et al. [6] suggests that clustering would be effective in CodeWrite as well.

Techniques for finding misconceptions are part of concept inventory generation [1, 4, 9, 10, 20]. However, these established techniques place a significant burden on experts and are therefore better-suited for general topics (such as algorithms and data structures) than for niche assignments. By contrast, Quizius [17] and PeerWise [5] are two lightweight tools for finding potential misconceptions without, e.g., conducting interviews. Like our method, both rely on data from learners and offer learners some benefits in return. We have used the name "classsourcing" to describe this virtuous relationship; an alternative name is learnersourcing [13].

[2] The work is exempted from our Institutional Review process, but we took reasonable safeguards.

8 DISCUSSION

When learners misunderstand a programming problem, instructors must determine what went wrong and provide actionable feedback. This paper contributes a classsourcing method that addresses the first half of this challenge—identifying and diagnosing misconceptions—by analyzing incorrect examples as part of a class-wide dataset. Our results show that the method can be used to improve the quality of feedback between course offerings by revealing expert blind spots. Across three assignments in two domains, programming and formal methods, we identified 26 potential misconceptions by sampling student examples.

These findings are especially encouraging because, contrary to what the paper might suggest, the course staff did not design chaffs in a vacuum, simply imagining what might go wrong. Rather, they used concrete input from students, such as what they noticed when grading homework submissions and what students asked questions about during office hours (personal communication). Despite this, classsourcing wheat failures still found notable improvements!

In addition, the method draws attention to "expert-only" chaffs that have no support from the dataset. Such chaffs may fail to reflect common mistakes, as we saw in the buggy symmetry chaff for U-Tree (section 5.1), but they may also be false alarms. A chaff may correspond to a real misconception that by chance did not show up in the data, and a chaff may intentionally add constraints that go beyond what the wheats require. Manual review is categorize.

For future work, we plan to tighten the feedback loop. Instructors need to identify misconceptions as early as possible, ideally during the term and before the assignment deadline. Automating some aspects of the method would help achieve this goal. Furthermore, automation would enable tools for individualized feedback such as an IDE plugin that synthesizes chaffs in response to errors.

Another important future topic is clustering. Our work depends heavily on the ability to form clusters from wheat failures: as we see, our total number of wheat failures can be quite large and hence, without clustering, overwhelming or useless. Clustering is also very time-consuming. Therefore, ideally, we would want to automate this step. However, it is unclear that standard clustering techniques would work: they are either for domains (such as images) that are very different from input/output examples, or they use syntactic distance metrics. In contrast, we need semantic clustering: e.g., `median([2, 1, 3]) is 1` and `median([5, 4, 6]) is 4` potentially have the same (middle) misconception, but are syntactically unrelated. Furthermore, we will need different methods, and may have different tools, for each language: e.g., we could potentially use tracing for programs, while we may be able to employ the tools of logic for formal specifications.

We close with an interesting observation: although the method of this paper helps to avoid expert blind spots, we had to overcome a blind spot *of our own* to create it! For eight years, the designers of Examplar never thought to examine wheat failures. It took a fresh perspective from the first author to recognize their value. Blind spots are tough.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation grants DGE-2208731, SHF-2227863, and 2030859 to the CRA for the CIFellows project. This work was also partially supported by RelationalAI. Thanks to Yanyan Ren, Kuang-Chen Lu, and Elijah Rivera for testing an early version of our talk-aloud study. Thanks to Shaun Wallace for pointers to the literature on crowdsourcing. Last but not least, thanks to the students in our classes and the participants in the talk-aloud.

REFERENCES

[1] Vicki L. Almstrum, Peter B. Henderson, Valerie J. Harvey, Cinda Heeren, William A. Marion, Charles Riedesel, Leen-Kiat Soh, and Allison Elliott Tew. 2006. Concept Inventories in Computer Science for the Topic Discrete Mathematics. *ACM SIGCSE Bulletin* 38, 4 (2006), 132–145. https://doi.org/10.1145/1189136.1189182

[2] Michael K. Bradshaw. 2015. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *SIGCSE*. 488–493. https://doi.org/10.1145/2676723.2677247

[3] Forge Contributors. 2022. *Forge: A Tool and Language for Teaching Formal Methods.* https://forge-fm.org Accessed 2022-09-21.

[4] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *SIGCSE*. 21–26. https://doi.org/10.1145/2157136.2157148

[5] Paul Denny, John Hamer, Andrew Luxton-Reilly, and Helen C. Purchase. 2008. PeerWise: Students Sharing their Multiple Choice Questions. In *ICER*. 51–58. https://doi.org/10.1145/1404520.1404526

[6] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Koli Calling*. 11:1–11:10. https://doi.org/10.1145/3364510.3366170

[7] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *ACM Journal on Educational Resources in Computing* 3, 3 (2003), 1:1–1:24. https://doi.org/10.1145/1029994.1029995

[8] B. Glaser and A. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Sociology Press.

[9] Kenneth J. Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa C. Kaczmarczyk, Michael C. Loui, and Craig B. Zilles. 2008. Identifying Important and Difficult Concepts in Introductory Computing Courses using a Delphi Process. In *SIGCSE*. 256–260. https://doi.org/10.1145/1352135.1352226

[10] Geoffrey L. Herman, Michael C. Loui, and Craig B. Zilles. 2010. Creating the Digital Logic Concept Inventory. In *SIGCSE*. 102–106. https://doi.org/10.1145/1734263.1734298

[11] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis* (2 ed.). MIT Press.

[12] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *CACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[13] Juho Kim. 2015. *Improving Learning with Collective Learner Activity.* Ph.D. Dissertation. Massachusetts Institute of Technology. https://hdl.handle.net/1721.1/101464

[14] Dastyni Loksa and Amy J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *ICER*. 83–91. https://doi.org/10.1145/2960310.2960334

[15] Will Marrero and Amber Settle. 2005. Testing First: Emphasizing Testing in Early Programming Courses. In *SIGCSE*. 4–8. https://doi.org/10.1145/1067445.1067451

[16] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani L. Peters, John Homer, and Maxine S. Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *ICER*. 41–50. https://doi.org/10.1145/3230977.3230981

[17] Sam Saarinen, Shriram Krishnamurthi, Kathi Fisler, and Preston Tunnell Wilson. 2019. Harnessing the Wisdom of the Classes: Classsourcing and Machine Learning for Assessment Instrument Generation. In *SIGCSE*. 606–612. https://doi.org/10.1145/3287324.3287504

[18] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *CACM* 18, 11 (1975), 613–620. https://doi.org/10.1145/361219.361220

[19] Abigail Siegel, Mia Santomauro, Tristan Dyer, Tim Nelson, and Shriram Krishnamurthi. 2021. Prototyping Formal Methods Tools: A Protocol Analysis Case Study. In *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday.* 394–413. https://doi.org/10.1007/978-3-030-91631-2_22

[20] Allison Elliott Tew and Mark Guzdial. 2010. Developing a Validated Assessment of Fundamental CS1 Concepts. In *SIGCSE*. 97–101. https://doi.org/10.1145/1734263.1734297

[21] Jacqueline L. Whalley and Nadia Kasto. 2014. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *ITiCSE*. 279–284. https://doi.org/10.1145/2591708.2591762

[22] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *ICER*. 131–139. https://doi.org/10.1145/3291279.3339416

[23] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *ICER*. 51–59. https://doi.org/10.1145/3230977.3230999

[24] John Sinclair Wrenn. 2022. *Executable Examples: Empowering Students to Hone Their Problem Comprehension.* Ph.D. Dissertation. Brown University. https://repository.library.brown.edu/studio/item/bdr:wgtu3pq6/