Will Students Write Tests Early Without Coercion?*

John Wrenn
Computer Science Department
Brown University
Providence, Rhode Island, USA
jswrenn@cs.brown.edu

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

ABSTRACT

Students faced with a programming task often begin their implementation without a sufficient understanding of the problem. Several prior papers suggest that formulating input–output examples *before* beginning one's implementation is the key to averting problem misunderstandings, but that students are loath to actually do it. Is outright *coercion* instructors' only hope to convince students to follow this methodology and hence help themselves?

We conjecture that students' reluctance may stem from the disaffordances of their programming environments. In this work, we augment the student's programming environment to encourage examples-first development, and design a novel measure to assess students' adherence to this methodology. We apply these measures to students using our modified environment in a semester-long course, and find high *voluntary* adherence, especially relative to the literature's low expectations.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; • **Social and professional topics → Student assessment**; CS1; • **Human-centered computing → User studies**.

KEYWORDS

Pyret, Examplar, CS1, examples-first, testing, self-regulation

ACM Reference Format:
John Wrenn and Shriram Krishnamurthi. 2020. Will Students Write Tests Early Without Coercion?. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20), November 19–22, 2020, Koli, Finland.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3428029.3428060

1 INTRODUCTION

Students faced with a programming problem often begin their implementation work without a complete understanding of the problem [7, 11, 12], because they lack the metacognitive awareness to self-regulate their progress [7]. Educators have long attempted to support metacognition by instructing students in an explicit problem solving methodology, ranging from Pólya's 1945 *How to Solve It* [9] for mathematics, to *How to Design Programs*'s "Design Recipe" [4], to Loksa et. al's 2016 six-step metacognitive scaffold [8]. All three of these scaffolds ask students to begin by *reinterpreting* the problem to ensure they understand it, and to end by reviewing their solution. In the Design Recipe, reinterpretation culminates with the development of illustrative input–output examples; review culminates in testing. (In our work, examples and tests share the same syntax; they differ only in their purpose.)

In controlled environments where students can be repeatedly reminded to follow these scaffolds, students have higher productivity, self-efficacy, and independence [8]. Where students are forced to solve input–output examples before beginning their implementations, they may produce better solutions [10] and make fewer errors [2]. However, left to their own devices students may lack the metacognitive awareness to realize they even need to apply these scaffolds. Students who are encouraged to follow the Design Recipe in lecture may not formulate *any* examples or test cases and consequently struggle [5]. Requiring the final submission of test cases on assignments *will* coax students to write them, but not necessarily well: A study by Edwards and Shams [3] of students trained in test-driven development and graded on test suite coverage found that students' tests were both few and uninformative; most students wrote exactly as many tests as there were methods, and those tests tended to only evaluate the "happy path" of their respective methods.

Given the bleak research literature, it is unsurprising that recent work [2, 10] has explored (with success) *forcing* students to solve input–output examples before allowing them to begin their implementation work. Unfortunately, forcing students can make them resentful of the activity, want to get out of it as quickly as possible, find it inauthentic, and create barriers of trust between students and faculty. Thus, coercion should only be a last resort.

Fortunately, there is also cause to not abandon hope of students' ability to *self*-regulate: our prior work demonstrates that students *can* produce numerous, high-quality tests [14] given the right assessment methodology, and that, when aided by helpful feedback, will even write tests when not required to do so by the assignment [13]. With the right incentives and support, students *can* be coaxed to test—but can they be coaxed to test *early*? In this work, we augment the student's programming environment to encourage examples-first development, and design a novel measure to assess students' adherence to this methodology. We apply these measures to students who used our modified environment in a semester-long accelerated introductory computer science course, and find high *voluntary* adherence, especially relative to the literature's low expectations.

*This work was supported in part by the US National Science Foundation.

2 PEDAGOGIC CONTEXT

We assess the submissions of roughly sixty students in an accelerated introductory computer science course at a private university in the US. Most were first-year students but many had prior computing experience, and all had to pass a set of programming exercises to gain entry. Enrollment declined slightly over the course of the semester: 64 students submitted the first assignment; 59 submitted the final assignment. Students were encouraged, but not required, to follow the Design Recipe—unless they sought help, at which point course staff would expect to see all the steps and help with the earliest incomplete one.

2.1 Assignment Structure

The course was project-oriented, featuring 17 programming projects.[1] For each of these assignments, students were required to submit a `code` file containing their implementation, a `tests` file containing their test suite, and a `common` file. The `common` file was a shared dependency of both the `tests` and `code` file and provided a place for course staff to provide common definitions, and for students to write definitions useful to both their `tests` and `code` files (e.g., a helper function).

Students' final test suites were primarily graded on the basis of their *validity* and *thoroughness* [13]. The *validity* of a test suite is a binary measure of whether it does or does not conform to the problem specification. A test suite is valid if it will accept all correct implementations of the problem. The *thoroughness* of a test suite is a measure of how effective the test cases are at catching bugs. It is computed by running the student's test suite against a set of known-buggy implementations, and calculating the proportion that the test suite rejected.

2.2 Programming Environment

For these assignments, students used a variant (pictured in fig. 1) of the standard Pyret programming environment, modified to nudge students towards the programming methodology urged by the instructor, i.e., to first explore the problem by writing interesting examples *before* their implementation efforts. Adherence to this methodology is a feat of self-regulation: a student must have the willingness to follow this advice, the control to deliberately delay implementation, and the perception to generate *interesting* examples. With four key modifications to students' IDE, we sought to nudge students to write examples early:

Testing is the Default. In the usual Pyret IDE, beginning one's implementation is effortless. In our modified IDE, the `code` file is *hidden* until students click a **Begin Implementation** button.

Fast File Switching: The standard, web-based Pyret IDE is a single-file editor; context switching between testing and implementing is slow. The modified IDE allows students to quickly switch between their `code`, `tests`, and `common` files.

Integrated Test Results: The standard Pyret IDE only displays `tests` file test results when running in that file—not when running in a `code` file that imports those tests. Our modified IDE shows test results on every run, *regardless* of the current file.

[1]http://cs19.cs.brown.edu/2019/assignments.html

Example Feedback: Our prior Examplar [13] IDE provided students with on-demand feedback on the validity and thoroughness of their examples, independent of their implementation progress. Unfortunately, it is a distinct tool from the IDE in which students do their implementation work; Examplar thus could only benefit students when they possessed the self-awareness to realize they ought to use it.

In contrast, our new IDE provides students with Examplar-style feedback on *every* run, regardless of the open file. As with our prior work [13], we compute the thoroughness aspect of this feedback using only a handful of buggy implementations that explore the *logically*-interesting aspects of the problem by realizing those aspects incorrectly (rather than subtle implementation bugs). This integrated feedback reifies the "interestingness" of a student's examples: each additional buggy implementation caught by the student provides an assurance that they correctly and thoroughly understand the problem.

3 PRIOR ART

Kazerouni et al. [6] propose a family of metrics to assess the *balance* and *ordering* of students' testing efforts and implementation efforts of students writing Java. In these metrics, "effort" is quantified by the number of line-level changes between file-saves (as measured by `git diff`), and is reported at three levels of granularity: *project*, *work-session*, and *method*.

Unfortunately, line-based metrics are inherently sensitive to syntactic quirks. Minor stylistic preferences between students may be reflected as substantial differences in effort. For instance, where one student might write:

```
let foo = if a { b } else { c };
```

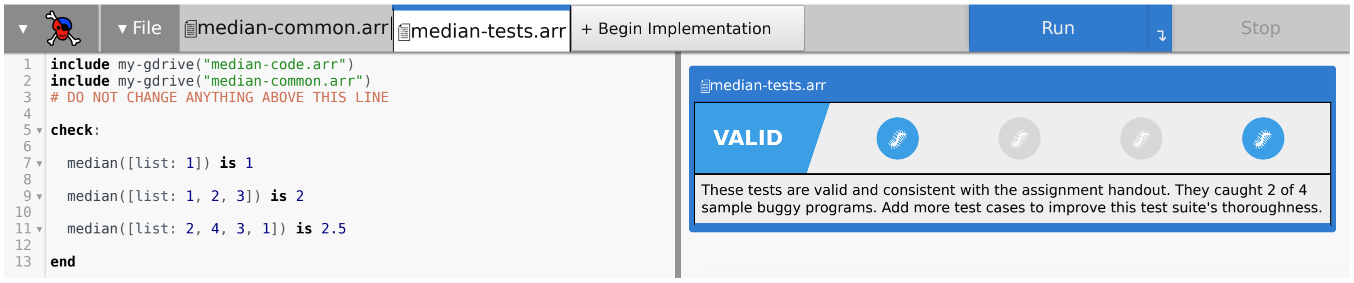
...another might, *equivalently*, write:

```
let foo;
if a {
  foo = b;
} else {
  foo = c;
}
```

It seems unlikely that the latter takes *six times* as much effort to write as the former.

Stylistic differences between students aside, line-based metrics may also misrepresent an individual student's balance of work between testing and implementation. In JUnit (the testing framework used by students in Kazerouni et al.'s work), the syntactic forms associated with implementation work are very similar to those associated with tests. However, some languages (like Pyret, which we use) have concise testing syntax, making "lines" incomparable.

Buffardi & Edwards [1] assessed students' adherence to test-driven development by computing each student's test coverage (against their own implementation) averaged across their submissions to an external automated assessment system. Unfortunately, if submitting to the automated assessment system is tedious, students may leave their IDE to do so only infrequently–this interval of observation is thus potentially too infrequent to compose an adequate picture of a student's incremental progress.



```
include my-gdrive("median-code.arr")
include my-gdrive("median-common.arr")
# DO NOT CHANGE ANYTHING ABOVE THIS LINE

check:

  median([list: 1]) is 1

  median([list: 1, 2, 3]) is 2

  median([list: 2, 4, 3, 1]) is 2.5

end
```

These tests are valid and consistent with the assignment handout. They caught 2 of 4 sample buggy programs. Add more test cases to improve this test suite's thoroughness.

Figure 1: Our editing environment requires students click Begin Implementation before beginning their implementation, and provides integrated feedback about the quality of tests on each run.

4 DISCRETIZING EFFORT

To assess students' testing and implementation efforts over time, we must define both an interval of observation, and a unit of effort. We derive both of these from clicks of the Run button.

The intervals created by successive runs provide a meaningful unit of effort: a programmer clicks Run with the frequency at which they wish to receive feedback. Whatever amount of testing or implementation a student completes between two successive runs reflects, by definition, the amount of testing or implementation effort which that student was comfortable undertaking on their own before requesting feedback again. In contrast, file saves only reflect the frequency at which students wish to preserve their work.

To quantify a student's balance of implementation versus testing effort, we might simply contrast the number of test-and-run intervals to the number of implement-and-run intervals. However, if students tend to both implement *and* test within the same run-intervals, these intervals will be too coarse to be informative: test-and-run intervals and implement-and-run intervals will be one-and-the-same. We posit that students tend to compartmentalize their work inside run-intervals to either testing or implementation—rarely both.

4.1 Compartmentalization of Effort

We therefore begin by asking: do students tend to compartmentalize their work-between-runs to one of either testing or implementation? **Yes.** To produce this answer, we instrumented our editor to track the files modified within each run interval, and then counted and compared the number of runs occurring after each of the eight possible combinations of file modifications:

$$|\{\}| = 7111 \quad \Big\} \text{ 6.45\% of runs followed no modifications}$$

$$\left.\begin{aligned} |\{code\}| &= 54026 \\ |\{tests\}| &= 34087 \\ |\{common\}| &= 4148 \end{aligned}\right\} \text{ 83.73\% of runs followed modifying one file}$$

$$\left.\begin{aligned} |\{code, tests\}| &= 5480 \\ |\{code, common\}| &= 1745 \\ |\{tests, common\}| &= 2191 \\ |\{code, tests, common\}| &= 1403 \end{aligned}\right\} \text{ 9.82\% of runs followed modifying multiple files}$$

Run-intervals in which students edited multiple files were rare: among the 98,932 run-intervals which included edits to students' `text` or code files, only 6,883 (6.96%) entailed edits to *both*.

4.2 Effort Across Assignments

How did effort vary between assignments? To assess this, we should not assume that effort, as measured by sheer number of run-intervals, is directly comparable between students; different students might simply tend to click Run with different frequency.

We begin by establishing, for each student, baselines which characterizes their "usual" number of testing and implementation run-intervals: that student's average number of implementation and testing intervals across all assignments, and the standard deviation of those quantities for each assignment. We then plot, in fig. 2, each student's relative quantity of implementation and testing intervals for each assignment, measured in units of standard deviations from that student's mean. (We exclude two assignments for which we failed to log data, and the three partner assignments.)

The testing and implementation effort involved in each assignment seems substantially impacted by the character of the assignment. For instance, the three assignments (Sortacle, Oracle, and MST) in which students implemented testing oracles (programs that test other programs) uniformly involved fewer-than-typical implementation and testing intervals. (A possible factor: these assignments do not have a binary notion of correctness, and thus integrated validity–thoroughness feedback was not available for them.) Another trio of similar assignments, TweeSearch1, TweeSearch2, and TweeSearch3, involved similar distributions of effort.

5 EXAMPLES-FIRST ADHERENCE

5.1 When do students click Begin Implementation?

Oftentimes immediately. Of 703 logged run sequences in which students clicked Begin Implementation,[2] 363 (51.64%) clicked Begin Implementation *before* clicking Run for the first time. Of these, 165 (45.45%) actually edited their code file within that same initial run-interval. This suggests that about half of students who clicked Begin Implementation immediately did *not* necessarily do so to begin their implementation.

Without a population to compare to, we cannot say with any certainty that hiding the code file behind a Begin Implementation button swayed students towards initial testing. Nonetheless, we are encouraged by these data. We anticipated that most students would

[2]There were 4 logged sequences in which students *never* clicked Begin Implementation. Three of these sequences occurred on assignments completed with a partner. We presume that, in these cases, the partner did the implementation work.

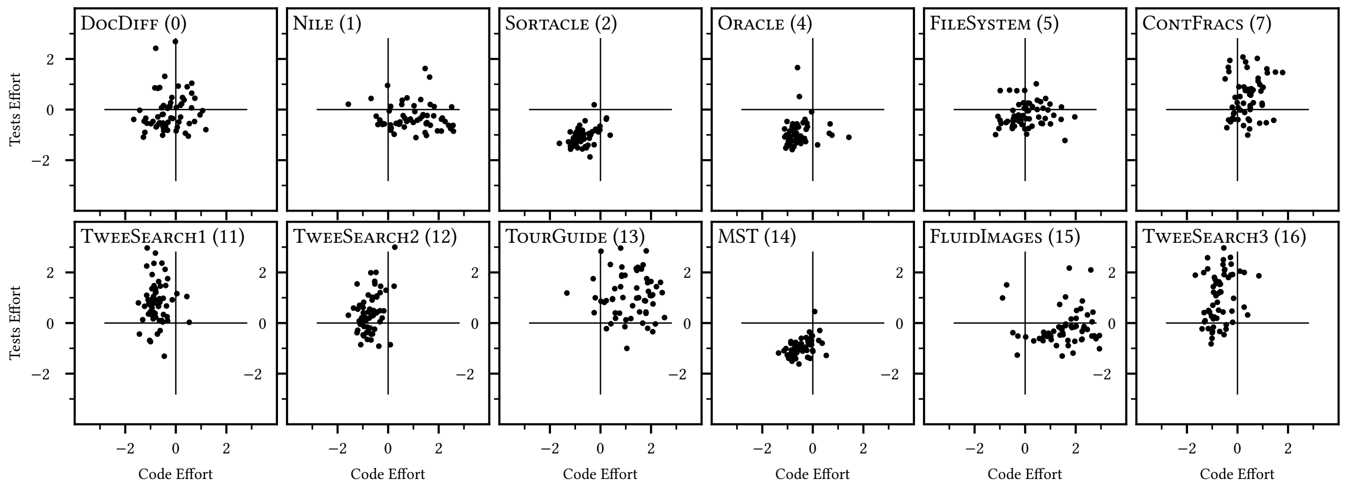


Figure 2: Each point reflects the relative testing and implementation effort of a student, measured in standard deviations from their typical testing and implementation efforts. Implementation effort ranges on the x-axes; testing ranges on the y-axes. Assignments are parenthetically numbered with the index of their appearance in the course.

reflexively click Begin Implementation immediately, yet about half of students did not. Of those who did, about half did not actually edit their implementation within that run-interval. This suggests that many students may be clicking Begin Implementation just to survey the initial contents of the code file.

These data point to a possible design improvement: if students wish to be able to see the code file, make its contents initially visible but un-editable (until students click Begin Implementation).

5.2 How thoroughly do students test prior to their implementation efforts?

A student who adheres to an examples-first programming methodology will develop *interesting* examples *prior* to their implementation efforts. The examples must be interesting, because uninteresting assertions do not probe one's understanding of the problem. This example-writing must occur prior to implementation, because writing input–output assertions *after* implementation is merely testing—tests *confirm* implementation correctness; examples *anticipate* it. Consequently, a good metric of examples-first adherence should:

- Evaluate the *quality* of examples—not the quantity: the measure should not reward uninteresting assertions, and should be unaffected by the volume of edits to the tests file.
- Reward a student's authorship of interesting examples *prior* to implementation: interesting examples written after implementation contribute less to problem understanding.
- Not penalize students for using tests to *review* their implementation efforts: modifications to `tests` after implementation efforts should not contribute negatively.

These properties are satisfied by the *mean implementation-interval thoroughness* (miit, for short): the mean of the peak thoroughness achieved prior to each implementation interval. Concretely, consider this (synthetic) sequence of run-intervals:

$$\left[\blacksquare^{1/5}, \blacksquare^{2/5}, \blacksquare^{2/5}, \square^{2/5}, \square^{❗}, \square^{2/5}, \blacksquare^{3/5}, \square^{3/5}, \blacksquare^{4/5}, \blacksquare^{5/5} \right]$$

The fractions denote the thoroughness feedback resulting from that run, and ❗ denotes that the run resulted in an error; ■ denotes modifications to `tests`, and □ denotes modifications to `code`. Now, consider *only* the impl(ementation)-intervals:

$$\left[\square^{2/5}, \square^{❗}, \square^{2/5}, \square^{3/5} \right]$$

This student completed 3 impl-intervals after achieving a peak thoroughness of 2/5, and one more after achieving a peak thoroughness of 3/5. Their miit is therefore $(2/5 \times 3 + 3/5 \times 1)/4 = 0.45$.

We use the *peak* thoroughness achieved prior to each implementation interval (as opposed to the *last* thoroughness achieved) because where thoroughness declines, it is usually because the student has written a very large test suite and has commented most or all of it out to either focus on a particular test result, or to temporarily hasten their edit-and-run cycle.

Figure 3 visualizes students' miit for each of the non-partner assignments with validity–thoroughness feedback. (We exclude the partner assignments, because we cannot always combine the logs of cooperating students into a single consistent timeline of work.) We hoped that providing students with integrated feedback on their examples would guide them to achieve some level of thoroughness prior to their implementation efforts. However, mindful of the frustration that ensues when students are unable to catch *all* the buggy implementations, the instructor told them to try to catch "most of them" but move on once they had done so instead of getting bogged down. Indeed, students typically *voluntarily* achieved a moderate level of thoroughness before the bulk of their implementation work: the miit of the median student ranged from 2/5 to 5/6. Only a handful

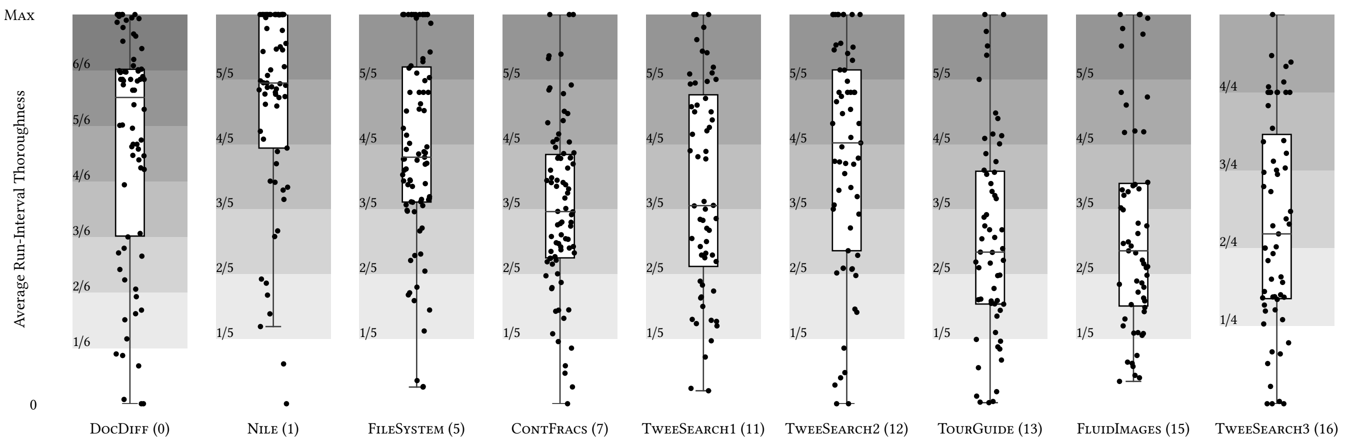


Figure 3: The MIIT of each student (represented by points) on each assignment. The shaded areas with fractional labels reflect each possible level of thoroughness as the proportion of buggy implementations rejected. Underlaid box-and-whiskers plots summarize the overall MIIT on each assignment.

of students on each assignment (six, on average) did not achieve any thoroughness before the bulk of their implementation work.

6 DISCUSSION

We did not observe in our students many of the bleak results logged by prior work: we found that most students did *not* begin their implementation work immediately, and that the vast majority wrote moderately thorough examples before their implementation efforts.

Absent further study, we cannot causally link our IDE modifications, specifically, to our positive results. Consider, for instance:

- Examples-first development is less rigid than test-driven development (which prescribes a strict interleaving of testing and implementation) and thus perhaps easier to adhere to.
- Pyret has a very lightweight, native syntax for test cases.
- The functions implemented by students are "pure", and thus more easily testable than if they involved side-effects.

Still, our work serves as a preliminary (positive) assessment of students' self-regulation abilities, as an example of how researchers might consider whether IDE disaffordances have warped student behavior, and as a toolbox for future analyses of students' example-writing behavior.

REFERENCES

[1] Kevin Buffardi and Stephen H. Edwards. 2013. Impacts of Adaptive Feedback on Teaching Test-Driven Development. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 293–298. https://doi.org/10.1145/2445196.2445287

[2] 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 *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3364510.3366170

[3] Stephen H. Edwards and Zalia Shams. 2014. Do Student Programmers All Tend to Write the Same Software Tests?. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. ACM, New York, NY, USA, 171–176. https://doi.org/10.1145/2591708.2591757

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs* (first ed.). MIT Press, Cambridge, MA, USA. http://www.htdp.org/

[5] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 12–20. https://doi.org/10.1145/3105726.3106183

[6] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 407–413. https://doi.org/10.1145/3287324.3287366

[7] Dastyni Loksa and Andrew J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. ACM, New York, NY, USA, 83–91. https://doi.org/10.1145/2960310.2960334

[8] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[9] George Pólya. 1945. *How to Solve it: A New Aspect of Mathematical Method.* Princeton University Press.

[10] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[11] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/3230977.3230981

[12] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. ACM, New York, NY, USA, 279–284. https://doi.org/10.1145/2591708.2591762

[13] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 131–139. https://doi.org/10.1145/3291279.3339416

[14] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999