Executable Examples for Programming Problem Comprehension

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

Flawed problem comprehension leads students to produce flawed implementations. However, testing alone is inadequate for checking comprehension: if a student develops both their tests and implementation with the same misunderstanding, running their tests against their implementation will not reveal the issue. As a solution, some pedagogies encourage the creation of input–output examples independent of testing—but seldom provide students with any mechanism to check that their examples are correct and thorough.

We propose a mechanism that provides students with instant feedback on their examples, independent of their implementation progress. We assess the impact of such an interface on an introductory programming course and find several positive impacts, some more neutral outcomes, and no identified negative effects.

KEYWORDS

Examplar, examples, testing, automated assessment

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION

In early algebra lessons, it is commonplace to challenge students to complete input–output tables that correspond to function specifications [18, 27]. In exploring these input–output examples, students check that their understanding of a function matches its actual behavior [18]. Software tests—which are usually also articulated as input–output pairs—play a similar role in computing. However, software testing is inadequate for checking problem comprehension: if a student develops both their tests and implementation with the same misunderstanding of a problem, running those tests against their implementation will not reveal their misunderstanding.

Flawed implementations often stem from underlying misunderstandings (section 2.1) and some pedagogies (section 2.2) attempt to address this by encouraging students to develop examples in the form of input–output assertions, independent of testing their implementations. However, without an implementation to run assertions against, examples are impotent and do not provide feedback. Consequently, students may be inclined to begin their implementations prematurely (a process whose ample feedback may mask underlying misunderstandings and instill a false sense of progress [26]).

Educators stressing the development of examples must therefore provide students with some mechanism to assess their understanding. In this work, we present such a mechanism. Examplar is a tool that provides students with instant feedback on whether they have correctly and thoroughly explored a problem independent of their implementation progress. In the presence of this interface, we ask: Did students in an accelerated introductory course...

rg 1: …choose to use Examplar?
rg 2: …ultimately submit more or better test cases?
rg 3: …ultimately submit more correct implementations?

2 THEORETICAL BASIS

2.1 Failures of Comprehension

Although comprehension is ubiquitously recognized as an indispensable component of problem solving, the computing education literature is rife with studies in which student participants inadvertently make significant progress solving the “wrong” problems:

Whalley and Kasto (ITiCSE ’14) [33]:

Interestingly, [three students] retrieved the 'counting integers' schema. The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

Loksa and Ko (ICER ’16) [21]:

Of all 37 participants, only 15 verbalized about reinterpretating the prompt. This lack of reinterpretation was consistent across both experience groups: [CS1 and CS2]. Participants often began coding without fully understanding the problem, leaving them with knowledge gaps in the problem requirements and causing them to later stop implementation to address the gaps.

Prather et al. (ICER ’18) [26]:

The most frequent issue these students encountered was a failure to build a correct conceptual model of the problem. The feedback from Athene seems to have given [several participants] a false sense of progression through the problem. Furthermore, there are no measures between viewing the problem and submitting source code to ensure that the student understands what they’re being asked to do.
2.2 Systematic Problem Solving

A potential way to improve students’ development of problem comprehension is to train them in a methodology that explicitly scaffolds the process. Our pedagogical context is the Design Recipe from How to Design Programs (HTDP) [11], a six-step process (summarized in fig. 1) for producing an implementation from a specification, grounded in multiple theoretical foundations. The first three of these steps specifically scaffold the development of problem comprehension, and its last step prompts students to confirm that their understanding matches their implementation.

At a high level, its steps provide a form of scaffolding [3] to lead a student from a prose-based problem statement to a working program. The scaffolding steps ask students to produce intermediate artifacts (signature/purpose, examples, code template) that capture the problem at multiple levels of detail and abstraction. The progression from data definitions to examples to code moves the student through different representations of the problem, providing a form of concreteness fading [16] as students progress towards a symbolic-form solution to a problem.

Completed sequences of design-recipe steps form worked examples [32] that students can leverage when considering new problems. A student might refer to a design recipe example when writing a new program on an already-studied datatype; this would focus on the examples, templates, and code features of the example. When asked to work with a new datatype, the recipe suggests higher-level steps that a student can follow to make progress on the problem.

Several papers have begun to explore the positive impact of the HTDP recipe on students in different contexts. Fisher and colleagues on multiple projects [13, 15] showed that HTDP-trained students made fewer programming errors than students trained in more conventional curricula. Schanzer et al. [29, 30] have found improvements in middle- and high-school students’ abilities to solve algebra word problems after working with a version of the design recipe.

However, students trained to follow HTDP may not formulate any examples or test their programs, and consequently struggle [14]. Furthermore, HTDP does not include any inherent mechanism for students to assess their own examples; examples are completely impotent until the student completes their implementation (step 5), at which point they become the basis of a test suite (step 6). Students may begin their implementations prematurely (at the expense of problem comprehension) because the implementation phase of problem solving provides feedback to a degree which comprehension development does not [26]. A mechanism to assess examples would enable students to check their comprehension (thus supporting a type of self-regulation [21]) and might prevent the tendency to implement prematurely.

3 ASSESSING TESTS & EXAMPLES

Input–output examples, like test cases, can be articulated as assertions of the input–output behavior of functions. To assess whether examples are valid and thorough explorations of a problem, we adapt the classifier perspective of assessing test suites [2, 24]. This view is appropriate for assignments consisting solely of deterministic, computable functions for which correctness is a well-defined binary property. We discuss how these expectations limit Examplar’s applicability in section 9. In this method, we assess suites of assertions along two axes: validity and thoroughness. However, unlike test cases, the intent of examples is not to test one’s implementation, but rather one’s understanding of the problem; we adapt our assessment of thoroughness to reflect this difference.

3.1 Validity

A suite is valid if it accepts (i.e., its assertions pass) all correct implementations. A suite may be invalid for a variety of reasons; particularly, it may have:

(1) asserted incorrect behavior (e.g., sorting in the wrong direction),
(2) asserted underspecified behavior (e.g., asserting that a sorting implementation is stable, if that was not specified),
(3) simply have failed to compile or run altogether.

We assess whether a suite is valid by running it against a particular, representative correct implementation. However, if the problem specification leaves any behavior underspecified, it is necessary to run suites against multiple correct implementations in order to accurately identify invalidity [35]. For instance, consider a problem specification that reads:

Write a function, median, that consumes a list of numbers and produces the arithmetic median.

This specification, as worded, leaves the behavior of median on empty inputs underspecified; it may be just as correct for an implementation to produce an error as to return 0. In order for a suite to be valid for all implementations of median, it must not include any assertions involving empty input lists. We can accurately identify such assertions as invalid by checking them against two correct implementations (henceforth wheats [24]):

(1) one that produces an error on empty inputs,
(2) another that produces some answer (say 0) on empty inputs.

If a student asserts that implementations should produce an error on empty inputs, their suite will reject the wheat that produces 0 (and vice versa). Provided that the set of wheats completely exercises the space of underspecified behaviors permitted by the specification, accepting all wheats guarantees that a suite is valid and will accept all correct implementations.
3.2 Thoroughness

A suite is thorough if it rejects (i.e., its assertions do not pass) buggy implementations. We assess the thoroughness of a suite by running it against a curated set of buggy implementations (henceforth chaffs [24]). The thoroughness of a suite is measured as the proportion of chaffs it rejects. To assess test suites, the set of chaffs should include subtly buggy implementations. To assess examples, we take a different perspective: the set of chaffs should exercise logical misunderstandings that students are likely to make. For instance, to assess the thoroughness of examples for median, the set of chaffs could include implementations of mean and mode.

4 EXAMPLAR

Examplar (pictured in fig. 2) provides a specialized version of the usual Pyret [5] editing environment1 tuned for writing examples as input–output assertions. Students write their assertions just as they would in Pyret’s usual editor. However, Examplar replaces the usual editor’s Run button with a Run Tests button, which assesses the student’s suite for validity and thoroughness against instructor-authored implementations (in the manner described in section 3). Consequently, students can use Examplar to develop and assess their examples independent of their implementation progress.

Pyret’s usual development environment provides extensive information in its presentation of errors and testing results2 [34]. For instance, if a test fails because the two halves of an equality assertion are not equal, Pyret displays the values that each half evaluated to. This is undesirable in Examplar, as a student may be overly tempted to intentionally write failing assertions to discover what the behavior of wheats is, rather than closely read the assignment specification to determine the behavior on their own. Our intention is that Examplar supplements—but does not replace—the assignment specification. Examplar therefore removes the interaction pane and suppresses nearly all forms of program output. Examplar only displays errors that prevent assertions from running.

5 RELATED WORK

Prior work has attempted to incentivize software testing with on-demand feedback. Stephen Edwards has conducted extensive research on the classroom integration of test-driven development (TDD) since 2003, specifically involving the automatic assessment tool Web-CAT [8]. While we fundamentally share Edwards’s view that testing can move developers from “from trial-and-error to reflection-in-action” [7], our approach differs in key ways. Web-CAT’s pedagogical context is TDD, in which the developer strictly interleaves testing with implementation. This may tempt students to write minimal tests in order to begin coding [10].

In Edwards’s work, the feedback which students receive on the quality of their tests is typically in terms of code coverage [6, 20]. The coverage of a test suite is ostensibly a measure of how effective the suite is at catching bugs. This measure is attractive because it reflects professional software engineering practice [22] and is not labor-intensive [6]. However, coverage is at best an indirect measure, since it does not involve observing whether a test suite actually catches bugs. Additionally, a growing body of evidence (including recent work from Edwards [4, 9]) challenges the assumption that coverage correlates with the thoroughness of a test suite [1, 19]. Examplar directly measures the quality of test suites by running them against actual buggy implementations (as described in section 3).

Prather et al. [25] ask students, before they begin their implementation, to correctly predict the output of the specified function for a given input. As with Examplar, this intervention provides an opportunity for students to verify that their understanding of the problem matches the prompt. However, Examplar differs from Prather et al.’s work in several key ways. Examplar requires that students develop their own input data for examples. Second, in addition to being valid, Examplar-assessed examples must also be thorough explorations of the problem’s interesting facets. Third, our students were welcome to use Examplar at any point in their development process (or not at all); Prather et al.’s intervention was strictly situated between reading the problem prompt and developing a solution.

6 METHOD

We deployed Examplar in fall 2018 in an accelerated introduction to computer science course offered at a selective, private U.S. university. The course instructs students on the design recipe, algorithm and data structure design, and algorithm (big-O) analysis.

Course Structure. The primary course activity was programming projects. The 2018 offering of the course featured fourteen programming projects. For all of these projects, students were given a prose specification and were required to submit an implementation consistent with that specification. For twelve of these projects, students additionally submitted a test suite. We provided Examplar on the ten projects that met the expectations outlined in section 3. The projects included constructing a recommendation engine, modeling a filesystem [12], deriving Huet zippers [17], and seam carving [31].

Demographics. Sixty-seven students completed the course. Most were first-year students, approximately 18 years old, with some prior programming experience (though not typically with prior testing experience). About 1/6 identified as female. Admittance to the course required successful completion of four assignments roughly corresponding to the first fifth of HTDP.

Pedagogy. The instructor asked students to follow the entirety of the design recipe while completing all programming projects. However, in requiring the submission of only a final implementation and a test suite, the course essentially enforced only the last two steps of the design recipe.

Previous iterations of the course attempted to apply the idea of a “sweep” [23]: graded examples due several days before the final submission deadline. The fast pacing of programming projects precluded this requirement for most assignments, but it was hoped the habit of early example-writing would stick. However, from the guilty admissions of former students,3 we believed that for projects lacking this early deadline, students authored most (if not all) of their assertions after developing their implementation. We hoped Examplar would be an effective alternative to strict early deadlines.

---

1https://code.pyret.org/editor

3In particular, the former students who were hired to be 2018’s TA staff!
6.1 **RQ 1: Do students use Examplar?**

To determine whether students used Examplar, we monitored their use of the tool, instrumenting Examplar to log the username and suite contents each time a user clicked *Run Tests*.

**RQ 1.1: ...when it is not required?** We wanted to see the degree to which students used Examplar on their own volition; we thus did not force students to use the tool. However, we feared students might not try the tool at all merely as a matter of lack of exposure. We therefore required that students use Examplar for the first assignment, but made usage optional thereafter. To judge whether students valued Examplar on their own volition, we compare submission volume for this first assignment to that of subsequent assignments.

**RQ 1.2: ...when no final test suite is required?** On one assignment, DataScripting, students were *not* required to submit a test suite, but Examplar was still provided. This assignment was a collection of seven, small, independent programming problems (adapted from Fisler et al. [15]). Students submitted independent implementation files for each part, and we provided independent Examplar instances for six of the seven parts. To judge whether students used Examplar when no final test suite was required, we compare volume of submissions for this assignment to that of the other assignments.

**RQ 1.3: ...throughout their development process?** The Examplar usage logs provide only a partial view of students’ overall development process; students still needed to use Pyret’s usual editing environment to develop their implementations and to run their test suite against their own implementations. Instrumenting Pyret’s usual editor was not feasible. The usage logs therefore do not tell us how students used Examplar in relation to their other development progress. We supplement our understanding with a voluntary survey prompting students for feedback to “help us evaluate if and how we use Examplar in future semesters”. In this survey, we asked students to self-report their use of Examplar, relative to their progress in developing their implementations.

6.2 **Do final submissions change?**

To determine whether Examplar induced changes in students’ final submissions, we looked at the 2017 offering of the course as a point of comparison. Aside from the introduction of Examplar, the 2018 offering of the course was virtually unchanged from the 2017 offering. Of the fourteen programming projects in the 2018 offering, thirteen appeared in the 2017 offering. Both offerings used the same entry process, featured similar lectures (which were held at the same times), and provided similar resources for students. The student demographic was almost nearly identical (except with 76 students, resulting in more total implementations and tests).

Naturally, there were some changes; we did not restrict the 2018 course staff from correcting significant issues as they saw fit. Nevertheless, of the ten assignments for which Examplar was provided, seven were functionally identical to their 2017 offering (table 1); i.e., we are able to meaningfully assess the submissions for both years using identical wheats, chaffs, and test suites. We use subsets of these comparable assignments to judge whether the quality of students’ final test suites and implementations improved. In section 8 we discuss the limitations of this evaluation approach and why we did not perform a more tightly controlled study.

**RQ 2: Do test suites change?** Of the comparable assignments, five required the submission of a test suite. We use these five assignments...
We perform a two-sample t-test to determine if the average number of bugs per test suite is significantly different between years. We hypothesized that, by gamifying the testing experience, Examplar would induce students to write more tests. We perform a two-sample t-test to determine if the average number of test cases significantly differs between years.

**RQ 2.1: Does test suite size increase?** To determine whether final test suite size increased, we contrast the number of tests in suites from each year. We hypothesized that, by gamifying the testing experience, Examplar would induce students to write more tests. We perform a two-sample t-test to determine if the average number of test cases significantly differs between years.

**RQ 2.2: Does validity improve?** We hypothesized that, in aggregate, the validity of final test suites would improve significantly from 2017 to 2018. Examplar’s feedback on validity is complete; i.e., if a test suite accepts all of the wheats in Examplar, it will accept all of the wheats in the autograder used for final submissions. We sort the implementations for each year into the dichotomous categories of valid and invalid (section 3.1), and perform a $\chi^2$ test to determine if the proportion of valid test suites differ significantly.

**RQ 2.3: Does thoroughness decline?** The chaffs used to assess final test suites included both mistakes of logic and implementation errors. However, Examplar only included chaffs targeting the former, so it is conceivable that students could misinterpret catching all chaffs within Examplar as having “finished” their test suite. We therefore must check whether the thoroughness of students’ test suites declined. We compute the thoroughness of each final test suite (section 3.2) and, conditioned on observing a decrease in the proportion of chaffs caught between years, perform a $\chi^2$ test to determine if the difference is significant.

**RQ 3: Do implementations change?** We hypothesized that the direct aid provided by Examplar for test development would indirectly benefit students’ implementations. All seven of the comparable assignments required the submission of implementations. This set of assignments is comprised of 622 implementations from 2017, and 522 from 2018. We sort the implementations for each year into the dichotomous categories of correct and buggy using an instructor-authored test suite, and perform a $\chi^2$ test to determine if the proportion of correct implementations significantly differ.

### Table 1: The position and duration of the comparable assignments in each year.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Ordinal</th>
<th>Days</th>
<th>2017</th>
<th>Ordinal</th>
<th>Days</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocDiff</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nile</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataScripting</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FileSystem</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updater</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JoinLists</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapReduce</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Did you use Examplar to write examples or tests Before, During, and After completing your implementation? (Higher percentages are shaded darker.)

<table>
<thead>
<tr>
<th>Usage</th>
<th>Before</th>
<th>During</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rarely, or not at all</td>
<td>4.3%</td>
<td>4.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>A few times</td>
<td>21.7%</td>
<td>13.0%</td>
<td>13.0%</td>
</tr>
<tr>
<td>About half of the time</td>
<td>39.1%</td>
<td>17.4%</td>
<td>17.4%</td>
</tr>
<tr>
<td>Most of the time</td>
<td>21.7%</td>
<td>26.1%</td>
<td>17.4%</td>
</tr>
<tr>
<td>Almost always, or always</td>
<td>13.0%</td>
<td>39.1%</td>
<td>52.2%</td>
</tr>
<tr>
<td>Unsure</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

## 7 Results

We present our findings for each of the research questions stated in section 6.

### 7.1 RQ 1: Did students use Examplar?

Students used Examplar extensively on all assignments, clicking Run Tests a total of 26,211 times. Figure 3 illustrates the distribution of Examplar submissions per-student for each of the assignments where Examplar was provided.

**RQ 1.1: ...when it was not required?** Yes. Students used Examplar extensively even after the requirement to use it was dropped. The median Examplar-submissions-per-student for DocDiff of 22 (the first and only assignment for which Examplar use was required) was less than that of any other assignment. Only a small number of students elected to not use Examplar thereafter: 4 students on DataScripting, 3 on FileSystem, and 1 on Updater, MapReduce, TourGuide, and FluidImages.

**RQ 1.2: ...when no final test suite was required?** Yes. Figure 4 illustrates the distribution of the number of Examplar submissions per-student for each of DataScripting’s parts. Of 67 students who submitted an implementation for at least one part, 64 used it for at least one part and 48 used it for every part for which they submitted an implementation. Examplar usage for this assignment is particularly notable as students were given only two days to complete its seven parts. Interpreted as a whole, Examplar usage for this assignment was on par with that for the other assignments; the median student submitted 33 suites to Examplar for DataScripting.

**RQ 1.3: ...throughout their development process?** Probably. Twenty-three students (approximately a third of the students enrolled in the course) provided feedback on their Examplar usage in the voluntary course feedback survey. When asked, “Did you use Examplar [before, during, after] developing your implementation?”, a majority of students indicated they used Examplar at least “about half the time” at all stages. Self-reported Examplar usage (table 2) increased as implementation development progressed. Of course, students’ self appraisal of their own testing diligence should be regarded with some skepticism, especially on a non-anonymous survey distributed a month after the course ended.

---

The individual parts of DataScripting received fewer submissions-per-student than DocDiff, but each was a significantly smaller problem than any other in the course.
Figure 3: For each assignment: a combined violin and box-and-whiskers plot illustrating the volume of Examplar-submissions-per-student for each assignment. The center line in each box represents the number of times the median student clicked Run Tests. The whiskers extend to the most extreme data lying within 1.5 times the interquartile range. Points indicate the exact number of submissions of outlying students.

Figure 4: Examplar submissions per-student for each of DataScripting parts, rendered in the same manner as fig. 3.

Figure 5: The number of assertions in final test suites from each year, rendered in the same manner as fig. 3.

7.2 Did final submissions change?

Yes. In aggregate, the quality of both test suites and implementations improved from 2017 to 2018.

**rq 2:** Did test suites change? Yes. In aggregate, the validity of final test suites significantly improved, without any degradation in their thoroughness, on the five comparable assignments. Curiously, there was no significant difference in the size of students’ test suites. We consider the size and quality of these suites in turn:

**rq 2.1: Did size increase?** No, the number of assertions in final test suites was approximately equal (fig. 5). With a Welch’s t-test, we determined that the difference in average size between suites in 2017 and 2018 was not significant ($t(523.79) = 0.66871, p = 0.504$).

**rq 2.2:** Did validity improve? Yes. In aggregate, final test suites in 2017 were 4.8 times more likely to be invalid than suites in 2018. Table 3 illustrates the proportion of invalid test suites for each of the comparable assignments. Our $\chi^2$ test with Yates’ continuity correction revealed that the validity of test suites on comparable assignments significantly differed by year ($\chi^2(1, N = 589) = 52.373, p < 0.01, \phi = 0.303$, the odds ratio is 0.16).

Why were fewer final test suites invalid in 2018? As discussed in section 3.1, there are three important causes of invalidity:

1. If a suite accepts some—but not all—wheats, it is almost certainly asserting underspecified behavior. Final test suites in 2017 were 12.9 times more likely to have this form of invalidity than suites in 2018.
we do not perform a $\chi^2$ test.

Table 4: For each of the comparable assignments and in aggregate: the number of final test suites of each form of invalidity for the comparable assignments in 2017 and 2018.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017</th>
<th>2018</th>
<th>2017</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Invalid $n$</td>
<td>Invalid $n$</td>
<td>Invalid $n$</td>
<td>Invalid $n$</td>
</tr>
<tr>
<td>DocDiff</td>
<td>29.7%</td>
<td>30.3%</td>
<td>622</td>
<td>632</td>
</tr>
<tr>
<td>Filesystem</td>
<td>30.3%</td>
<td>30.3%</td>
<td>622</td>
<td>632</td>
</tr>
<tr>
<td>Updater</td>
<td>20.0%</td>
<td>28.0%</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>JoinLists</td>
<td>17.9%</td>
<td>17.9%</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>MapReduce</td>
<td>64.1%</td>
<td>64.1%</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Aggregate</td>
<td>30.3%</td>
<td>30.3%</td>
<td>269</td>
<td>269</td>
</tr>
</tbody>
</table>

Table 5: For each of the comparable assignments and in aggregate: the number of chaffs used to assess students’ final test suites, and the proportion of final chaffs caught, on average, by students’ final test suites.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Chaffs</th>
<th>% Final Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Examplar Final</td>
<td>2017</td>
</tr>
<tr>
<td>DocDiff</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Filesystem</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Updater</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>JoinLists</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>MapReduce</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Aggregate</td>
<td>26</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 6: For each of the comparable assignments and in aggregate: the proportion of the final implementation submissions for each year that were correct.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correct $n$</td>
<td>Correct $n$</td>
</tr>
<tr>
<td>DocDiff</td>
<td>63.7%</td>
<td>74.7%</td>
</tr>
<tr>
<td>Filesystem</td>
<td>59.5%</td>
<td>68.6%</td>
</tr>
<tr>
<td>AddingMachine</td>
<td>38.2%</td>
<td>54.1%</td>
</tr>
<tr>
<td>Palindrome</td>
<td>86.8%</td>
<td>88.5%</td>
</tr>
<tr>
<td>SumLargest</td>
<td>84.2%</td>
<td>91.8%</td>
</tr>
<tr>
<td>Filesystem</td>
<td>77.6%</td>
<td>62.9%</td>
</tr>
<tr>
<td>Updater</td>
<td>28.0%</td>
<td>36.4%</td>
</tr>
<tr>
<td>JoinLists</td>
<td>64.1%</td>
<td>75.8%</td>
</tr>
<tr>
<td>MapReduce</td>
<td>69.2%</td>
<td>63.6%</td>
</tr>
<tr>
<td>Aggregate</td>
<td>63.2%</td>
<td>68.2%</td>
</tr>
</tbody>
</table>
9 THREATS TO GENERALIZABILITY

There are a handful of situations in which Examplar is poorly suited:

**Correctness:** In the classifier approach to test suite assessment, correctness must be a well-defined, binary property of implementations. We could not provide Examplar for two assignments which lacked this property. For instance, in SORTACLE, students implemented the function `sortacle`, which consumed a sorting function and produced `true` if the function was correct (and `false`, otherwise) by checking it against a large number of generated inputs. A quality `sortacle` will be very good at labeling sorts accurately, but this is an impossible task to do perfectly: it is always possible to craft a sorting function so deviously buggy that no `sortacle` will detect the flaw. Since it is impossible to craft a true wheat for such an assignment, the classifier approach is inappropriate.

**Expressibility:** Examplar is not useful when articulating examples accurately is difficult or impossible. For instance, TourGuide asked students to implement a function that consumes a graph of locations, start points and end points, and produces the length of the shortest path between those termini. Unfortunately, the Euclidian distances between points is very often *irrational* and therefore must be represented approximately in floating point. To complicate matters, the accuracy of a summation of floating point numbers depends on the order in which the numbers are added. Thus, it is very difficult to express the right answer accurately on a computer.

**Determinism:** Examplar can be used for assignments that allow for an element of non-determinism, a form of underspecified behavior. However, if the wheats and chaffs loaded into Examplar are non-deterministic, Examplar may provide students with differing feedback between runs of the same suite. The allowance of non-determinism should therefore be realized by loading Examplar with multiple wheats that differ in their behavior, but are individually deterministic.

10 INDIRECT BENEFITS

In addition to the quantifiable improvements we observed in students’ final submissions, Examplar provided a host of other benefits:

**Reduced Load on Course Staff.** On the assignments for which we could not provide Examplar, course staffers reported an uptick in questions that they felt could have been resolved by Examplar; this also shows up in data gathered about the use of hours [28, §6.1.2].

**More Robust Autograding.** Providing Examplar instances forced the course staff to finalize the wheats for each assignment before the assignment went out to students. This process uncovered major issues in four assignments, before they were released to students.

**Teaching Underspecified Behavior.** Underspecification was not a learning goal of the course. However, it appears that some students *did* gain an understanding of what underspecified behavior is via their use of Examplar. We received several Piazza posts in which students discovered they were testing underspecified behavior, e.g.:

> My test below was rejected by a wheat. I think this might because I’m checking for unspecified behavior i.e. when an empty string is passed into the content of a file. Are we meant to assume that an empty string can never be passed into the content of a file?

11 FUTURE WORK

We briefly discuss two essential directions for future work:

**Over-Incentivation?** Students seemed to enjoy Examplar’s gamification immensely (e.g., fig. 6). Yet, on one assignment, Filesystem, the proportion of correct implementations declined precipitously in 2018. We attribute this decline to differences in the time allotted: students in 2018 were allotted half as much time as students in 2017. Nevertheless, both the validity and thoroughness of test suites for this assignment improved in 2018. We believe Examplar may have monopolized students’ time with test development—benefiting their test suites, but to the detriment of their implementations.

We can adjust Examplar’s “game” via our selection of chaffs, but this needs experimentation. Too few, and students may prematurely conclude that they are “done” with testing (and, crucially, problem comprehension). Too many, and students may divert too much time to Examplar and too little towards developing their implementation. Finding this balance is therefore essential future work.

**Unaccounted Factors.** Edwards and Shams [10] recently characterized a corpus of student-authored test suites as being short (only one student wrote more than 21 test cases), similar (89% of students wrote exactly 21 test cases), and ineffective (their test suites missed a “significant proportion” of bugs). However, the test suites we reviewed (produced both with and without the aid of Examplar) were generally long (students wrote an average of 39 test cases), varied significantly in length (some students wrote more than 200 test cases), and were highly effective at catching bugs.

This contrast leads us to believe there are manifold unaccounted factors that significantly affect students’ ability to write tests. The suites we studied were produced in an environment differing from Edwards and Shams in population, course level, prior experience, language, problems, pedagogy, and tooling. A holistic understanding of these factors is essential to moving our understanding of testing pedagogy out of its infancy—but our significantly different outcomes should provide an incentive for doing so.

ACKNOWLEDGMENTS

We thank the attendees of ICER ’18, who spurred this work by arguing that it would not succeed; Yanan Ren, who aided with wheats and chaffs; Asha Misra and Rebecca Aman, who proofread the paper, and tipped us off to Examplar’s social media debut; and Hero Park for agreeing to be named. This work was supported in part by the US National Science Foundation.