The Sweep: Essential Examples for In-Flow Peer Review

Joe Gibbs Politz† Joseph M. Collard◦ Arjun Guha◦ Kathi Fisler‡ Shriram Krishnamurthi†
†Brown University ◦UMass Amherst ‡WPI
sk@cs.brown.edu

ABSTRACT

In in-flow peer review, students provide feedback to one another on intermediate artifacts on their way to a final submission. Prior work has studied examples and tests as a potentially useful initial artifact for review. Unfortunately, large test suites are onerous to produce and especially to review. We instead propose the notion of a *sweep*, an artificially constrained set of tests that illustrates common and interesting behavior. We present experimental data across several courses that show that sweeps have reasonable quality, and are also a good target for peer review; for example, students usually (over half the time) suggest new tests to one another in a review.

1. EXAMPLES, TESTS, AND PEER REVIEW

Testing is an integral skill for students to learn, and writing tests before writing code is a theme in both industrial programming and in educational contexts. Some pedagogies, like *How to Design Programs* [6], make writing a few example invocations of a function a required step in writing programs, with exhaustive tests added as the function is fleshed out. Others teach the more rigid protocol of test-*driven* development [1], where students must write a test for each incremental piece of functionality before implementing it.

Seemingly orthogonal to testing, peer review is another important skill for students to learn. In-flow peer review (IFPR) [3] is a contributing student pedagogy [8] that integrates peer (code) review into assignments with low overhead. In particular, IFPR assignments have students submit parts of their work early on to be used as artifacts for peer review. Students can then use the feedback from review to improve their work before the final submission.

Prior work has combined these software engineering skills, using tests as an initial artifact [10]. Tests are an especially good review artifact: they can be read relatively independent of each other; they describe only expected behavior, before having to identify an implementation strategy; and they force the author and reader to consider all of program's input and output space. Reviewers can point out both *correctness* mistakes that misunderstand problem behavior, and *thoroughness* issues that don't cover important cases. IFPR has been used as early as first-year college courses.

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA

 ISBN 978-1-4503-3685-7/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2839509.2844626

2. SUITES AND SWEEPS

Following this other research, we have been experimenting with IFPR in courses that use a test-first (but not test-driven) style of program development. Students are required to submit test suites for review. As test suites, they are expected to comprehensively cover the problem space. Students receive feedback from others well in time to use it for their final submissions.

Unfortunately, from conversations with students (and from numerous complaints filed with TAs), it quickly became apparent that writing and reviewing a complete test suite in isolation was quite a difficult task, for three main reasons.

- We wanted students to submit tests early—usually two days into a week-long assignment. This is very onerous, and many tests only become clear after students have done some implementation, which they usually have not yet begun.
- While in principle unit tests should be written independent of an implementation, in practice it is hard to write them without being able to check for at least basic properties like type-correctness and well-formedness of values.
- Most of all, students were encouraged (by our grading system: section 3) to write comprehensive test suites, resulting in suites with hundreds of tests. These were onerous to review, leading to low-quality reviews and disaffected students.

Consequently, we decided to focus the initial submissions on artifacts that could be both produced and reviewed in a short amount of time, while still containing useful content. Concretely, we instructed students to initially submit 5–10 interesting, representative examples, dubbed the *sweep*,[1] rather than a comprehensive suite. This unburdens students from covering an entire input space, and lets them focus on exploring the interesting parts of the problem. Rather than exhaustively testing the program, they are instead coming up with representative *examples* that illustrate (and check!) important features of the program's behavior.

It is worth noting that the sweep idea has independent value. API documentation needs small sets of carefully-chosen examples. Exploring a new library for the first time involves coming up with interesting examples inputs to try. Separating feature-critical, first-priority tests from more in-depth system tests is a useful software engineering skill. Thus, the idea behind the sweep shows up in many later, professional contexts.[2]

[1] "Sweep" is a term from archaeology for an area containing a non-comprehensive, but interesting, survey of artifacts (https://en.wikipedia.org/wiki/Archaeological_field_survey#Fieldwalking_.28transects.29). We have also heard the term used in geology, to refer to a similar overview sample of rocks.

[2] In this paper: *tests* refer to comprehensive suites; *examples* are

Eventually—with the final submission—students must construct a strong test suite, going beyond the sweep. Shifting the focus of peer review from test suites to sweeps raises several questions, which we explore in this paper:

- Are sweeps sophisticated enough to cover an interesting part of the problem space, or does their brevity inhibit their value?
- Do students all test the same cases in their sweeps?
- Are sweeps (nearly) as comprehensive as full test suites, so full test suites were somewhat unnecessary all along?
- Do students give useful review feedback on sweeps?
- Do students share ideas about tests when reviewing sweeps?
- Are sweeps so simple that students don't make any errors for reviewers to point out, removing a useful kind of feedback?

We proceed by first discussing some of the mechanics of sweep-based assignments, including assessing test cases and the workflow of review (section 3), and the data we collected for this study (section 4.1). We then examine the behavior of student sweeps and suites for thoroughness and correctness (section 4.2). Finally, we explore the students' reviewing activity when giving feedback on one another's sweeps (section 4.3).

3. ASSESSING SUITES AND SWEEPS

When testing is an explicit part of an assignment, it ought to be assessed. There are a number of metrics we could use to do this.

One is code coverage – how much of the student's implementation is covered when running the test cases? For example-first approaches, this metric doesn't make much sense because the tests are written independent of the student's implementation! Once the student finishes the problem, including an implementation, coverage could be used to assess their tests, so the sweep could be held in reserve until the end of the assignment and assessed then. Two problems remain: first, it is difficult for a student to predict whether a particular sweep would cover all paths in a final implementation since that is dependent on a yet-unknown structure, and second, the sweep doesn't necessarily cover all cases by design, so grading strictly based on code coverage could be unfair.

Another technique is to run students' tests against one another's implementations, and give points for bugs caught [5]. It becomes necessary to avoid double-counting bugs caught if students, say, copy tests with minor permutations. One solution is to only count the number of other student solutions caught, rather than the number of tests that catch them. Again, this requires waiting until the end of the assignment to assess the sweep.

In section 1, we hinted at two metrics for evaluating tests:

- *Correctness* – do they have the right input/output behavior?
- *Thoroughness* – do the tests cover the input space?

Thoroughness is distinct from code coverage. It measures how thoroughly the *problem input space* is covered, not one particular implementation. For example, a function that uses division could have tests with perfect code coverage that still lack thoroughness, because they don't trigger a possible divide-by-zero exception.

Given these two metrics, how do we apply them to particular student tests? One way to evaluate correctness is to run the tests against a correct reference implementation to check that they pass, which we call a *gold* implementation. The more tests passed, the better the correctness of the suite or sweep. To check thoroughness, we can run the tests against several incorrect implementations or *coal*s, and check if the tests fail. If some test fails on a coal that didn't fail on the gold implementation, the we give the student credit for *detecting* that bad implementation. This gold/coal grading technique is useful for evaluating tests in general, and it is related to *mutation testing* [4], an existing technique in software engineering for ensuring the quality of a testing process.

When we introduced the sweep, students rightfully pointed out that it would be unfair to grade sweeps against the same standard as full test suites. To assess sweeps, we set some fraction of (often easier) coals as the number that needed to be caught for full credit on the sweep. The full set was used to assess final submissions.

Finally, human judgment is a useful way to assess tests: an expert can relatively quickly determine what problems could slip through a set of tests unobserved. In some testing cases, it's difficult to automate assessment with coals because the problem has multiple interacting pieces (so it's not clear what part to make the "coal"), or may involve visual components that aren't easy to automatically check. As a result, in some cases test cases were assessed manually by course staff, rather than through an automated process.

exploratory checks, usually written before and independent of the implementation; and *sweeps* are small example suites meant to capture interesting behavior rather than acheive complete coverage.

4. EFFECTIVENESS OF SWEEPS

The research questions in section 2 target the potential benefits of sweeps in peer review, as well as their potential limitations. Our formal analysis of those questions uses both quantitative and qualitative analyses: we use the gold/coal methodology outlined in section 3 to assess the correctness, thoroughness, and diversity of sweeps, and we manually analyze the content of reviews and review feedback to assess how students perceived the value of sweep-based peer review. Parts of these analyses explore sweeps independent of peer review, while others combine aspects of sweeps and peer review. Usually this will be clear from context, but we will also make this explicit where there might be doubt.

We intentionally do not attempt to compare peer review with sweeps to peer review with "full" test suites. A proper study of this form would require a controlled experiment, ideally with the same students doing sweeps on some assignments and not on others. The sweep arose because students found peer review with full test suites burdensome to the point that it was having adverse impacts on their experience in the course. Rather than (irresponsibly) prolong a bad pedagogic decision for the sake of a study, we instead chose to analyze sweeps on their own, letting our negative experience with the original model speak for itself.

4.1 Study Logistics

Courses and Population.

In this study, we use data collected from assignments in 3 courses across 2 institutions, both in the United States. Figure 1 summarizes the courses. CSPL is a programming languages course taught to sophomores through graduate students. CS1.5 is an accelerated introduction to data structures and algorithms for first-semester college students. CS2 is an introduction to data structures, algorithms, and object-oriented programming. The Assignments column lists the number of assignments per course that reviewed sweeps.

Assignment and Reviewing Workflow.

Assignment length varied from 4-5 days to two weeks, with most lasting one week. For week-long assignments, students submitted

Tag	Institution	Enrollment	Assignments
CSPL	I1	62	5
CS1.5	I1	60	7
CS2	I2	144	4

Figure 1: Courses with assignments involved in the study

sweeps by midnight on the second day. For example, an assignment released Friday and due the following Thursday midnight would have sweeps due by Sunday midnight. When assignment duration varied, the sweep deadline was roughly a third of the way through.

Students were explicitly instructed to submit 5-10 examples in the sweep. A single example was usually a single testing assertion (checking for two values being equal, for a predicate being true, etc.), but sometimes one logical example required multiple checks, so students submitted sweeps with more than 10 assertions.

All work was submitted through a Web interface that managed submissions, review assignment and authoring, and feedback. Upon submission, students were assigned other sweeps to review immediately if available, or put in a queue for review assignment if not. Sweeps were assigned to students until they had been reviewed 3 times each, and each student received 3 sweeps to review (TAs could fill in reviews for the last few submissions if needed).

Reviews followed a rubric that varied across courses and the semester, but asked roughly the same 3 questions:

- Are the tests correct?
- Do the tests cover an important and interesting set of cases?
- What did you like about the tests?

Students could browse code and mark incorrect tests with inline comments describing what was wrong, and give freeform feedback on the other questions. There were also Likert scales provided to quantitatively report on correctness and thoroughness.

Students could give two kinds of feedback on the review process. First, they could provide feedback on the helpfulness of individual reviews, including flagging them for inappropriate content (students didn't flag any reviews as hostile, though they did flag some for having negligible content). Then, after each assignment, students could fill out a survey indicating what changes they made both as a result of reviews they received and other sweeps they saw. Both kinds of feedback were visible only to the course staff.

Reviews *did not* count towards students' grades. The software we used wouldn't let students move on to their final submission without completing reviews, so they had to provide some content in the review form. We sent some messages via email (outside the system) to students who submitted reviews late to remind them of the review deadline, but this did not negatively affect their grade. Our experience thus reflects students' review habits unmotivated by any particular grading structure for them.

Altogether, students submitted around 1106 sweeps and 1975 reviews across the 16 sweep-based assignments. The rest of this section examines different parts of that corpus, starting with the sweeps themselves, before exploring the reviews.

4.2 Sweep Characteristics

We begin by analyzing the quality of sweeps themselves. For peer review to be valuable pedagogically, the artifacts to review need to contain issues for students to comment on, be deep enough

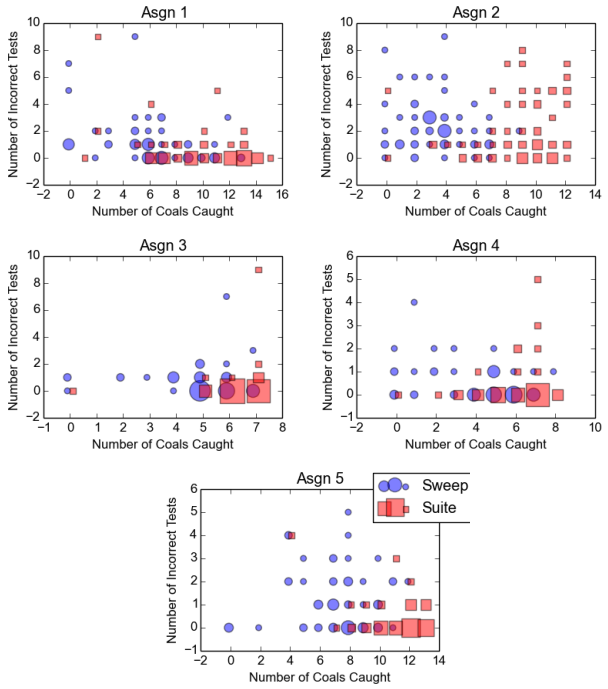


Figure 2: Sweep and suite test assessment

for students to find value in reading them, and be diverse enough that students don't all submit the same thing.

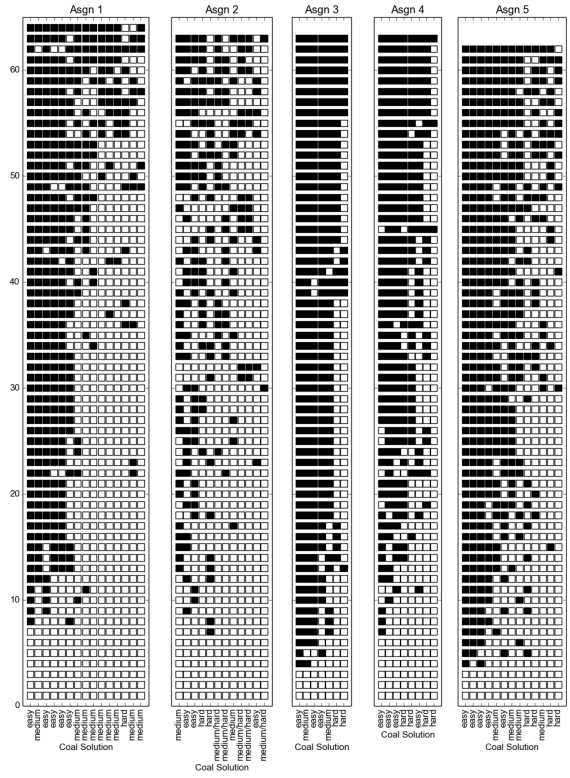
We limit our quantitative analysis (Section 4.2.1 and section 4.2.2) to assignments from CSPL, which had the most readily automatable gold/coal infrastructure. Our qualitative analysis (section 4.3) addresses peer review data from all courses.

4.2.1 Sweep Thoroughness and Correctness

CSPL staff wrote 6-16 coals for each assignment, depending on the complexity of the assignment and its amenability to coal implementation. The coals were intentionally designed to span a range of difficulty, with some containing obvious errors in simple base cases, and the hardest having subtle bugs in specific circumstances. Thus, we expected students to initially catch some easy coals, and eventually build a thorough suite to detect most or all of them.

Figure 2 summarizes both thoroughness and correctness of both sweeps and students' final test suites, with one plot for each assignment. Circles indicate sweep counts, and squares indicate final suite counts. Larger (proportionally) dots indicate more submissions were assessed with the values at that particular point. The x-axis shows the number of coals caught, so dots further to the right indicate more broad sets of tests. The y-axis shows the number of *incorrect* test cases, so dots closer to the bottom indicate more correct sets of tests. Thus, submissions at the bottom right do as well as possible by this metric, and those at the top left are the worst.

In these plots, we are most interested in whether sweeps contain errors (for reviewers to comment on) and cover multiple coals (suggesting depth in the sweeps, even though they should not be comprehensive by design). The plots largely show both in each assignment. We include the data on the final test suites to show that students' work progresses significantly beyond the sweeps, as one would hope. That final suites cover more coals and converge



■/□ – Coal caught/not caught by sweep (each row is a sweep submission)

Figure 3: Sweep variety measured by coal detection

on the lower right is neither surprising nor interesting. Comparing numbers of incorrect tests across sweeps and suites is tricky, given that the number of examples is limited in sweeps but not in suites. Overall, these data suggest that sweeps are sufficiently deep and error-prone that they are worthy of peer review.

4.2.2 Sweep Diversity

Next we explore whether sweeps are *diverse* enough across students to enable valuable peer review. We measure diversity as the distribution of coals detected by sweeps for the same assignment. The sweeps that caught each coal are shown for the same five assignments in figure 3. In each plot, rows represent submissions and columns represent coals. A square is filled if the sweep caught the coal, and empty if not. The submissions are ordered decreasing from top to bottom by number of coals detected, and coals are ordered decreasing from left to right by total number of students detecting each. The labels on the columns indicate the course staff's intent (before running anything) of how difficult each coal would be to detect. The layout allows us to observe a few things at a glance:

- In all 5 assignments, all coals were caught by some sweep, including the intentionally difficult ones. So students' sweeps cover the breadth of coals designed for assessment.
- In all assignments, there are some coals that are commonly caught by sweeps, and some that aren't. For example, the last "hard" coal on assignment 4 is caught by very few sweeps, while the first "easy" coal is caught by most.

Assignment	In	Out	Assignment	In	Out
CSPL/3	47	15	CS1.5/A	35	25
CSPL/4	41	21	CS1.5/B	39	21
CSPL/5	40	22	CS1.5/C	38	22
			CS1.5/D	37	23
CSPL/All	37	12	CS1.5/All	19	8
CSPL/Once	50	25	CS1.5/Once	52	41

Assignment numbers for CSPL are the same as in figure 2 and 3. Assignments 1 and 2 in CSPL did not allow students to opt out.

Figure 4: Student choice in assignments with opt-out

- All assignments show some variety in the sweeps as measured by coal, with some exhibiting much more than others. Assignments 3 and 4 show particularly little variety, and 1, 2, and 5 show more. We note that assignments 3 and 4 are also the assignments with fewer coals (the instructional staff reported that those problems were trickier to design interesting coals for). In these cases, we may be seeing the lack of diversity in coals more than the lack of variety in sweeps.

This analysis shows one measure of variety between sweeps, which is the different behaviors they exercise across coal solutions. There may be others; for example, the sweeps may use a different test structure or style that affords greater clarity, or they may detect the same coals with a more concise set of tests. However, just by the metric of coal detection, we see interesting variety across students in these assignments, which is a promising indicator of the value, in terms of test diversity, students could get when seeing one another's submissions during review.

4.3 Peer Review of Sweeps

Our quantitative analysis suggests that sweeps are artifacts worthy of peer-review: they have interesting content, they contain mistakes, and they provide a space of differences among students. We now turn to more qualitative analyses to determine whether sweeps are useful review artifacts in practice. We explore both students' impressions of sweep-based peer review and the content of the reviews themselves, checking whether students provide actionable recommendations in their reviews.

4.3.1 Student Opinion and Engagement

Since the semester for CSPL started with a non-sweep version of assignments, and we switched styles in part to respond to student feedback, we surveyed the students in that class to find out if they found the sweep helpful. We asked: *Since switching to the sweep (the new style of test submission), review has been helpful when I've participated, whether receiving or writing reviews.* 37 of 62 students responded, with 5 strongly agreeing, 17 agreeing, 10 slightly agreeing, 3 slightly disagreeing, and only 1 each disagreeing and strongly disagreeing. We took this as positive feedback that students were at least *perceiving* value from the process.

In addition, in both CSPL and CS1.5, students were given the option of opting out of sweep-based peer review a few assignments into the semester. This was mainly in response to concerns about the time burden of peer review. As a result, students could choose at sweep submission time on each assignment whether or not to participate in review. Students who opted out of review still had to

	CS1.5		CSPL		CS2		α
LGTM	16.8%	(74)	24.8%	(71)	45.4%	(163)	*
ERR	38.3%	(169)	20.3%	(58)	18.9%	(68)	.81
SPEC	33.6%	(148)	14.0%	(40)	12.5%	(45)	.55
TEST	58.0%	(256)	64.0%	(183)	42.1%	(151)	.88
NEG	1.4%	(6)	0.0%	(0)	2.2%	(8)	.00
TIP	28.6%	(126)	9.8%	(28)	11.1%	(40)	.72
REV	4.5%	(20)	4.5%	(13)	2.2%	(8)	.85
POS0	11.3%	(50)	23.1%	(66)	30.9%	(111)	.37
POS1	63.7%	(281)	62.6%	(179)	54.9%	(197)	
POS2	24.9%	(110)	14.3%	(41)	14.2%	(51)	

Figure 5: Results of coding reviews of sweeps

submit a sweep—because of our use of example-first programming practices—which was graded against the same standards; they only did not receive or provide reviews.

The results for students opting in are shown in figure 4. Between 30 and 40% of students opted out of review, which was roughly constant across both courses. Perhaps more remarkable is that 60–70% of students voluntarily participated in doing the extra work of peer review, which we believe argues for the (perceived) value of this process. The "All" row shows how many students made the choice to opt in or out consistently across all assignments, and "Once" shows how many students made the choice at least once. By this metric, in CSPL students were much more consistent with their choices than in CS1.5. We do not have a clear explanation for the difference between the two; in part it may just be that across four assignments in CS1.5 there was more opportunity for churn.

We do know that some students simply did not want to bother with an extra step in assignments, and were happy with their achievement already. It is tempting to believe that allowing students who aren't engaged to opt out may improve the reviews of all those who were motivated to participate. However, review improvement from opt-out isn't inevitable, as the students who opt out could actually the best reviewers, and removing them from the review pool would reduce the overall quality of review. For example, it's worth noting that CSPL has a more uniform cross-section of that department's population, while CS1.5 consists of students who placed into the accelerated course, and are thus more likely to view themselves as "hot shots". We leave the effects of student attitudes like these on review participation and quality for a future study.

4.3.2 Review Content Analysis

We manually analyzed a random sample of 1088 reviews from all the courses. Reviews can be valuable in many ways: they can point to errors in understanding the problem, suggest additional (concrete) examples, suggest changes to code style, or provide encouragement. Sometimes, reviewers gain insight from the work they are reviewing. In contrast, reviews that offer only cursory comments or are hostile devalue peer review. Our analysis looked for these positive and negative traits in reviews of sweeps.

A Rubric for Evaluating Reviews.

We developed a category-based rubric for coding reviews, iteratively checking inter-coder reliability and refining the categories. We settled on seven categories, which we summarize here:

- **(ERR) Identified (Potential) Mistake** – Did the reviewer claim to identify a programming mistake or incorrect testt?
- **(SPEC) Discussed or Disputed Problem Specification** – Did the reviewer remind the reviewee of correct behavior or point out an incorrect input/output pair? This can, but doesn't necessarily, overlap with ERR.
- **(TEST) Suggested Additional Test(s)/Identified Untested Feature** – Did the reviewer point out a specific gap in the thoroughness of the tests?
- **(POS) Complimentary/Positive (Scale of 0-2)** – Did the reviewer have specific praise for (parts of) the sweep?
- **(NEG) Hostile/Negative** – Was the review unnecessarily sarcastic, or blatantly hostile?
- **(TIP) Coding or Style Tip/Test Structure Comment** – Did the review give advice unrelated to the sweep or problem specification, like a coding trick or advice on test structure?
- **(REV) Reviewer Gained/Confirmed Understanding** – Did the reviewer indicate that they learned from performing the review, or confirm their own understanding?

Data Assessment and Rubric Agreement.

Figure 5 summarizes the results across the sample for these rubric categories, except for the the LGTM row, which we explain shortly. The last column reports Krippendorff's α calculated per category across 3 coders (each involved in at least one of the study courses) on 200 samples. Under this measure, strong agreement occurs at .8 and above, with tentative agreement above .67. As POS is scalar, its α score is aggregated in the row for POS0. When there was disagreement between coders in the 200 samples used for reliability testing, we counted the coding chosen by two of three coders.

The α column shows strong inter-coder agreement on three categories and no agreement on SPEC, POS, and NEG. We suspect that the low agreement on SPEC arose because the coders were not experts on all the assignments being coded; deciding if particular comments were about the assignment itself was very difficult. The POS category was highly subjective: coders had difficulty agreeing on the degree of positive tone in reviews. The NEG category has a 0 alpha score because there was only one case of a negative review labelled in the overlap, and the coders disagreed.

Observations on the Reviews.

The most common feature of reviews was a suggestion of a new test (category TEST): this happens in over half the reviews we sampled. Since sweeps are not comprehensive by design, we should expect that students *could* identify additional tests to suggest. This figure shows that students *do* share testing ideas through reviews. In addition, if we take a somewhat naïve understanding of the percentages in the categories and assume they happen uniformly, we could predict that roughly half of reviews will suggest a test to a student. That means that a student receiving just 2 reviews on a sweep ought to get some benefit (we have not analyzed this hypothesis).

The second most common feature is a reviewer claiming to identify out some kind of error (category ERR), whether a programming mistake or an input/output misunderstanding. This happened more rarely than suggesting an additional test, which could be for a number of reasons: errors are rarer because sweeps are short, or errors are simply harder to find than additional tests are to suggest (especially given the diversity of sweeps across students). Again, if we take a naïve interpretation, a student receiving 3 reviews ought to

expect to get an error reported most of the time (this is more clearly naïve than for additional tests, since students with no errors in their sweeps ought to not have errors pointed out to them). Nevertheless, any identification of purported errors is clear evidence of students attempting to perform a conscientious job of reviewing.

We gauged how often students received cursory (largely content-free but non-empty) reviews through an LGTM[3] category which we computed from categories in the rubric. We labelled a review LGTM if it had no other high-reliability category selected (we ignore SPEC, POS, and NEG since they aren't reliable). Since fewer than half the reviews are LGTM by our measure, this suggests that students getting multiple reviews are likely to get *some* non-vacuous feedback from peer review of their sweep.

There are other interesting observations across courses. First, the ordering of percentages are the same in all three courses—TEST, ERR, SPEC, TIP, REV, NEG—suggesting some cross-course consistency. In contrast, the magnitudes of the different categories differ quite a bit between courses. CS2 has the highest LGTM percentage and lowest percentage across the other categories. CS2 had a few major differences from the other two courses in the study that we could conjecture as explanations for the higher rate of LGTM reviews, and use as hypotheses in future studies. It (a) was a mandadory course, (b) had largest enrollment, and (c) didn't provide students with an opt-out for peer review. Finally, if we take these categories as indications of review utility, CS1.5 had the most useful reviews, which would fit an understanding that matches student motivation to review performance–CS1.5 is an advanced introductory course with prerequisite programming exercises required to enter. Further studies are required to probe at these questions of class size, review options, and motivation.

5. RELATED WORK

Testing and peer code review are well-studied topics in computer science education; we focus on work that emphasizes both, or is especially related to the example-driven style of the sweep.

Buffardi and Edwards study automated, rather than peer-written, feedback on test cases [2]. The feedback was different in three main ways: the style was test-driven, with students encouraged to maintain complete code coverage; the feedback took the form of hints on students' *solutions*, rather than tests; and the feedback was *earned* by submitting good tests. On different assignment styles (or at different stages of the same assignment), the two approaches could be complimentary.

Kulkarni, et al. study the impact of early exposure to examples in creative work, specifically drawings [9]. While their work is not programming-related, the sweep has similar goals in exposing students to diverse examples early on in the assignment process. Their work finds that when exposed to a variety of examples early on, subjects produced more varied final results in their own work, which is related to students suggesting new test ideas during review.

Smith, et al. don't do test-first programming, but do have students test one another's implementations [12]. Similarly, Reily, et al. use test cases as part of a code review feedback rubric [11]. These are different in that students are specifically testing an implementation, but similar in that tests and reviews are closely linked. A test in their workflows is a concrete piece of feedback that can be used to reproduce bugs in the other student's submission. Our process emphasizes generating ideas and understanding of a shared problem specification, while theirs is targeted at feedback on robustness and bug-finding in an existing implementation.

Gaspar, et al. report on Peer Testing, where students share test suites but do not review them [7], which was positively received by students. Since students do not produce reviews, the value comes from sharing tests, a value present in sweep-based assignments as well. This corresponds to the REV category in our coding rubric, where reviewers indicated that they learned something.

[3]An abbreviation for "Looks good to me". In some settings, it describes lazy reviews that do not indicate whether or not the reviewer actually analyzed the submission.

Acknowledgments.

This work is partially supported by the US National Science Foundation. We thank courses' staffs, especially Justin Pombrio.

Bibliography

[1] Kevin Buffardi and Stephen H. Edwards. Impacts of Teaching Test-Driven Development to Novice Programmers. *International Journal of Information and Computer Science*, 2012.

[2] Kevin Buffardi and Stephen H. Edwards. Impacts of Adaptive Feedback on Teaching Test-Driven Development. In *Proc. Special Interest Group on Computer Science Education*, 2013.

[3] Dave Clarke, Tony Clear, Kathi Fisler, Matthias Hauswirth, Shriram Krishnamurthi, Joe Gibbs Politz, Ville Tirronen, and Tobias Wrigstad. In-flow Peer Review. In *Proc. Innovation and Technology in Computer Science Education*, 2014.

[4] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEE Computer*, 1978.

[5] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. Running students' software tests against each others' code: New life for an old "gimmick". In *Proc. Special Interest Group on Computer Science Education*, 2012.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. How to Design Programs. MIT Press, 2002.

[7] Alessio Gaspar, Sarah Langevin, Naomi Boyer, and Ralph Tindell. A Preliminary Review of Undergraduate Programming Students' Perspectives on Writing Tests, Working with Others, & Using Peer Testing. In *Proc. Special Interest Group on Information Technology Education*, 2013.

[8] John Hamer, Quintin Cutts, Jana Jackova, Andrew Luxton-Reilly, Robert McCartney, Helen Purchase, Charles Riedesel, Mara Saeli, Kate Sanders, and Judithe Sheard. Contributing Student Pedagogy. *SIGCSE Bulletin* 40(4), pp. 194–212, 2008.

[9] Chinmay Kulkarni, Steven P. Dow, and Scott R Klemmer. Early and Repeated Exposure to Examples Improves Creative Work. In *Proc. Design Thinking Research*, 2014.

[10] Joe Gibbs Politz, Shriram Krishnamurthi, and Kathi Fisler. In-flow Peer Review of Tests in Test-First Programming. In *Proc. International Computing Education Research*, 2014.

[11] Ken Reily, Pam Ludford Finnerty, and Loren Terveen. Two peers are better than one: Aggregating peer reviews for computing assignments is surprisingly accurate. In *Proc. ACM International Conference on Supporting Group Work*, 2009.

[12] Joanna Smith, Joe Tessler, Elliot Kramer, and Calvin Lin. Using Peer Review to Teach Software Testing. In *Proc. International Computing Education Research*, 2012.