CaptainTeach: Multi-Stage, In-Flow Peer Review for Programming Assignments

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Daniel Patterson
Brown University
dbp@dbpmail.net

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Kathi Fisler
WPI
kfisler@cs.wpi.edu

ABSTRACT

Computing educators have used peer review in various ways in courses at many levels. Few of these efforts have applied peer review to multiple deliverables (such as specifications, tests, and code) within the same programming problem, or to assignments that are still in progress (as opposed to completed). This paper describes CaptainTeach, a programming environment enhanced with peer-review capabilities at multiple stages within assignments in progress. Multi-stage, in-flow peer review raises many logistical and pedagogical issues. This paper describes CaptainTeach and our experience using it in two undergraduate courses (one first-year and one upper-level); our analysis emphasizes issues that arise from the conjunction of multiple stages and in-flow reviewing, rather than peer review in general.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

Keywords

Peer-review, Learning environments, Testing

1. INTRODUCTION

Peer review has various educational roles, including encouraging reflection and metacognition [10], fostering critical-thinking skills [5], and assisting in producing feedback or grades in large courses [6, 8]. Many faculty have experimented with peer review within computing courses, both introductory and upper-level. The vast majority of these efforts have applied peer review (a) after the assignment has been turned in, and (b) to one deliverable within the assignment. There are, however, strong arguments for relaxing each of these constraints.

Reviews would ideally help students improve their work: just as students utilize office hours and discussion boards, peer reviews can provide valuable diagnostic information. This is perhaps especially true for programming assignments, whose problem specifications imply several objective criteria. If a student misunderstood a programming problem, peer-review while the assignment is open (henceforth *in-flow* reviewing) could help the student get back on track before the deadline. Naturally, this only works if students complete assignments early enough to receive and act on feedback. One way to accomplish this is to decompose assignments into multiple stages that each have concrete deliverables (e.g., requiring tests before code), and reviewing after each stage.

In Fall 2013, we experimented with multi-stage, in-flow peer review in two undergraduate courses (one a freshman honors course that emphasized data structures[1], the other an upper-level programming languages course[2]). We built an enhanced programming environment, CaptainTeach, to support the reviewing process. For the programming assignments in this paper, CaptainTeach used the programming language Pyret[3], whose design supports multi-stage reviewing (Section 3 discusses how).

CaptainTeach's novelty stems primarily from its integration of two ideas: (1) peer review and (2) decomposition of programming assignments into multiple deliverables that checkpoint student understanding of the assignment. Students submit and review others' work on each step before being allowed to submit work on subsequent steps. This paper focuses on logistical and policy considerations around multi-stage, in-flow reviewing for programming courses. Our evaluation focuses on whether this model is feasible and useful from a students' perspective. Section 6 contrasts CaptainTeach to other peer-review practices from the literature.

2. DECOMPOSING PROGRAMMING PROBLEMS FOR REVIEW

CaptainTeach strives to use peer review to help students adjust their work on an assignment before the due date. While reviewing complete drafts of programs is certainly possible, we'd like to help students catch their mistakes even earlier, before producing a complete program. In CaptainTeach, we design assignments to have sequential, reviewable deliverables and have students review one another's work at these intermediate stages. We explain how we decompose assignments for CaptainTeach through an example.

Consider this data-structure programming problem:

[1] `http://cs.brown.edu/courses/cs019/2013/`
[2] `http://cs.brown.edu/courses/cs173/2013/`
[3] `http://pyret.org/`

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 ACM 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.
ITiCSE '14 Uppsala, Sweden
Copyright 2014 ACM 978-1-4503-2833-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2591708.2591738.

> Write a program that takes a binary tree of numbers and produces a list containing those numbers according to a pre-order traversal of the tree.

Before a student can write a correct solution to this problem in any setting, she must be able to (1) develop and use a binary-tree data structure and (2) understand the term "pre-order traversal". In our experience, students who come to office hours with "code that doesn't work" are often stuck on one of these two more fundamental problems.

The *How to Design Programs* [4] methodology provides a generalizable "design recipe" for staging programming problems such as this. The full recipe asks students to approach a problem through seven ordered steps (paraphrased here for an experienced computer science audience; the presentation for beginners is worded differently):

1. Create the data structures needed for the input.
2. Create concrete instances of the data structure.
3. Write a type signature and summary of the function.
4. Write a set of test cases for the function, including the code needed to call the function on concrete inputs and the concrete answer that the function should produce (typically using the concrete data from step 1).
5. Write a skeleton of code that traverses the input data structure (but omits problem-specific logic).
6. Add problem-specific logic to the traversal skeleton.
7. Run the tests (from step 4) against the function.

Each step yields a concrete artifact that targets a different aspect of the problem; if a student is unable to produce one of the artifacts, he is likely to have trouble producing a correct and justifiable final program. When we work with *How to Design Programs* in person, we take students through each of these steps: a student asking for help must show each step before we will help with a later step (meaning we do not look at code—a step 6 concern—until the prior steps are done). In our long experience using this curriculum, many errors in student programs manifest in one of the early steps. Thus, the early steps are vital for making sure that students understand the problem.

In theory, one could build a programming environment that asks students to submit work for each step separately, optionally with reviewing on each step (though peer-review on all steps would almost certainly be too cumbersome). In our courses, we chose a coarser granularity. We provided the data structures (step 1) and function header (step 3), because having consistent shapes, field names, and function headers reduces confusion and enables one person's tests to probe another's code. Students submitted work twice per problem: in the first stage, they submitted black-box test cases (step 4) and any defined constants used in their test cases (step 2). In the second stage, they submitted their implementations (combining steps 5 and 6); we assumed that students ran their code against their own tests (step 7) before submitting. On one assignment, we used a three-stage process in which students also developed their own data structures (step 1) as the first stage.

The split in CaptainTeach of tests before implementations reflects software-engineering practices such as test-first development. In CaptainTeach, we use tests to checkpoint students' understanding of a problem: if a student misunderstands a question, misses a corner case, or misuses a data structure, his test cases usually reflect this error.

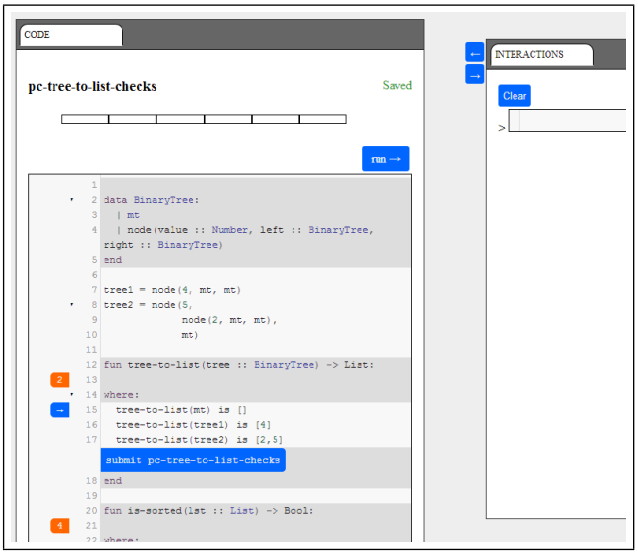

Figure 1: CaptainTeach enabled for writing tests.

3. CaptainTeach: THE TOOL AND ITS REVIEWING WORKFLOW

CaptainTeach typically takes students through a prescribed sequence of four core steps on each programming problem:

1. submit a test suite for review
2. review 2–3 other students' test suites
3. submit an implementation for review
4. review 2–3 other students' implementations

Students get reviews from others on their own work as each individual review is completed. Students can update their submissions until the assignment closes, but can only submit once for reviewing (to avoid over-burdening reviewers). Section 4 discusses grading of submissions. This section focuses on the interface, reviewing process, and language issues.

Figure 1 shows the interface when a student first starts a problem in CaptainTeach (through a web browser). In the right half of the page, students can evaluate expressions at a prompt: this is a standard interactive read-eval-print-loop (REPL). The left half of the page is the editing area, with the shaded portions locked to prevent editing. The data structure definition appears first (locked, so that everyone uses the same one), followed by an open area where students can define examples of the data structure. The next shaded area gives the name and type signature for the function. That area ends with a keyword `where`, which marks the start of test cases for the function. Students can enter their tests in the unshaded area that follows. Just under the test area is a button for submitting tests for review.

Markings in the leftmost portion of the editing area show students where they are in the sequence of steps. The arrow marks the area that students are currently expected to edit (though they may edit in any unshaded area). Numbered tags next to shaded areas show the order in which other areas will open for editing.

Once a student submits tests for review, her editing page gets additional tabs for others' work that she needs to review, as seen at the top of Figure 2. Her own code is still accessible in the CODE tab, but she will not be able to edit

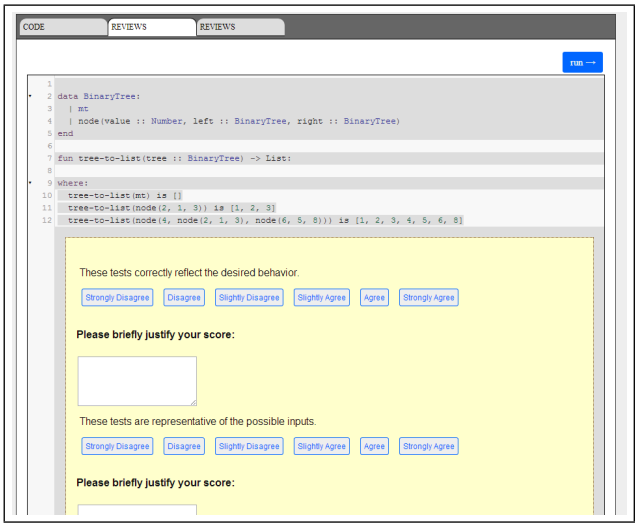

Figure 2: Interface for reviewing others' tests.

her implementation until after she submits reviews for each REVIEWS tab. The figure shows the review form for test cases. There are two prompts, each asking for a Likert-scale rating and free-form comments. One prompt asks about the correctness of the tests, while the other asks about test coverage. The button to submit the review has been truncated from the image, but appears below the lower prompt.

Once the student has submitted reviews for each tab, the area for her implementation unlocks in her CODE tab (the interface is identical to Figure 1 except the area marked with 2 is no longer shaded and the progress tags have changed). The review form for implementations also has two prompts, one about correctness and one about design and structure:

- "This code correctly implements the desired behavior."
- "This code is structured well."

Though CaptainTeach could be applied to any language, it works especially well with Pyret. In most languages, testing is a library feature so the environment must necessarily guess the loci of tests (which may even be split across files). In contrast, Pyret has linguistic support for testing in the form of `where` (and another kind of block indicated by `check`), so the environment can unambiguously identify where the tests should go, enabling it to lock and unlock portions of the editor, and also report test failures more accurately.

4. LOGISTICS AND POLICIES

Multi-stage, in-flow reviewing raises several logistical and policy questions. Some affect the design of CaptainTeach, while others impact course policies.

Whether to Synchronize Deadlines Per Stage.

With multi-stage reviewing, instructors can either allow students to complete stages at their own pace or synchronize the class through a separate due date per stage. We chose the former, so that peer-review would not constrain students' workflows. Section 5 reports timing patterns in how students submitted across stages.

Non-synchronized submission could block progress for students who submit first. We seeded CaptainTeach with several solutions, both good and bad, to give early-submitters something to review. In practice, multiple students submitted initial stages around the same times, so each students' work was sent out for review fairly quickly.

Grading and Reviews.

Other work on peer-review has unsurprisingly observed that students take reviewing more seriously when reviews themselves are graded or count towards course grades. We did not grade reviews, and used a combination of human TAs and results from an automated testing harness to determine actual grades for code submissions.

We motivated good reviewing in several ways:

- Students had to submit reviews in order to submit subsequent stages of the assignment or revisions to their own submission for the same stage.
- If students gave a bad review score to a good seed (or vice-versa), CaptainTeach told them so immediately. The seed solution and their review were accessible from their assignment page, so they could discuss their review with TAs if necessary.
- The recipient of a review could give *review feedback* to the reviewer, indicating if the review was helpful. This feedback consisted of both a Likert-scale rating and free-form comments.

The course staff also spot-checked reviews of seed solutions for general content and tone. This process never identified problems, so the staff used it only lightly later in the courses.

Potential for Plagiarism.

Showing students examples of others' work while an assignment is open might invite plagiarism. Students can copy code they are reviewing and paste it into their own solutions; also, nothing prevents students from helping one another by pasting their own solutions into reviews.

We could integrate code-similarity detection tools into CaptainTeach, but have chosen to not do so. Plagiarism has not historically been a problem in these courses, and they are small enough that human TAs can flag blatant plagiarism. More importantly, instead of fighting an uphill battle, we adopt the following two perspectives on plagiarism enabled by CaptainTeach:

First, we designed our grading policies to mitigate the effect of copying. For each deliverable (tests and implementation), students have two official submissions: one submitted for review and the final version (whatever was present when the assignment closed). Any influence from other students would reflect in the latter, but not the former. When grading assignments, the initial submission was weighted heavily (75% versus 25%): this allows students to benefit from reviewing insights, but does not allow someone to pass the course on the work of others. This also nudges students to submit what they believe is a quality submission before taking up valuable reviewing resources.

This ratio does hurt the student who does poorly at first, having significantly misunderstood the assignment, but uses reviewing as intended to correct their mistakes before the final submission. Having students submit tests first ameliorates this impact, as significant misconceptions about the problem should manifest in the test cases, before the student submits a correspondingly incorrect implementation. In addition, given that there were multiple course assignments, a

significant misconception on one or two assignments would be unlikely to adversely impact the overall course grade. Finally, these discrepancies are easy to notice when making up final grades.

Second, at a more principled level, we (controversially!) view code availability as a valuable learning opportunity for students, in that it reflects modern software practice. Sites such as StackOverflow and blogs give programmers ready access to code for various tasks; the problem for users is to assess whether the code they find is worth copying. CaptainTeach anonymizes the work being reviewed, so students cannot rely on the reputation of authors when deciding whether to follow ideas seen in other students' work; instead, they must judge the code itself. If they make a correct judgement, this demonstrates learning and a corresponding improvement in their grade. However, they may also copy a wrong solution (even one we used to seed the system, which can look quite convincing!), so copying blindly is perilous.

Disrupting Mental Flow.

The reviewing process interrupts students as they work on an assignment. This interruption could be helpful, as it potentially reveals errors before students begin implementation. The interruption could also be disruptive, as students have to wrap their heads around someone else's solution between phases of working on their own.

We suspect that being asked to review would be most disruptive in the midst of working on the program body. However, test-first approaches develop tests with no dependency on an implementation. This means that when reviewing test cases, students have just finished a standalone test-writing activity, and are presumably about to transition to solution strategies. Therefore, we hope the timing of the first review minimizes disruption and maximizes reflection, and we did not receive student complaints about it.

In feedback given during the semester, students raised one concern about the mental burden of reviewing: they wanted to review work from the same students across the multiple stages. Currently, CaptainTeach assigns reviews based on submission time, and does not retain reviewer-reviewee pairs across steps. Student felt they could reuse mental effort across the review stages had they been reviewing work from the same authors. The downside to this proposal lies in timing: students might receive feedback later if they have to wait for their original reviewers to submit work. This is an interesting design choice to explore in the future.

In a similar vein, students requested an additional round of back-and-forth with their reviewers to seek clarification on comments. We had considered but not implemented this out of concern that it would be too distracting and time-consuming. In an open discussion on the system, however, students felt that one additional *optional* round would have struck an appropriate balance.

5. EVALUATION

CaptainTeach presumes that peer-reviewing helps students get quick feedback on their work (both through reviews and through self-assessment after seeing the work of others). To understand how the multi-stage aspect of CaptainTeach worked in practice, we analyzed the timings at which students completed each stage relative to the overall assignment due date, as well as student opinions on the relative value of reviewing across the stages.

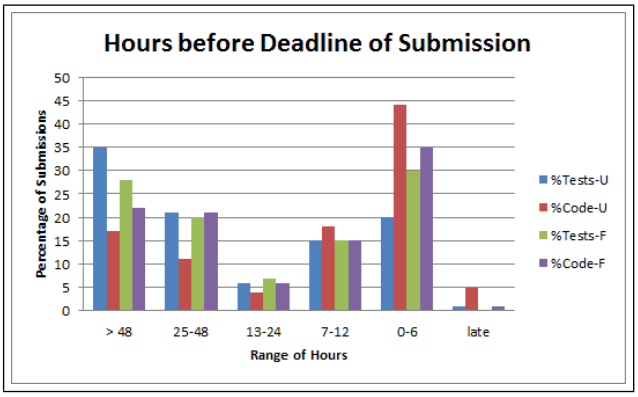

Figure 3: Bars show the percentage of submissions made within the range of hours on the x-axis. "U" and "F" in the legend refer to the upper-level and freshman course, respectively.

The freshman course used CaptainTeach on 4 assignments, with 2, 4, 6, and 17 reviewed steps. The upper-level course used it on 8 programming assignments, each with 2 reviewed steps, and 5 written assignments, each with a single reviewed step. For the 2-step assignments, the steps were (1) tests for, then (2) implementation of a single function. Assignments with 4 (resp. 6) steps, had 2 (resp. 3) sequential instances of writing tests for, then implementation of, a single function. For the 17-step assignment, students first wrote a data definition, and then wrote 8 functions in tests-then-code style.[4]

5.1 Submission and Reviewing Behavior

Figure 3 shows how far in advance students submitted each of tests and code in each course. With the exception of code in the upper-level course, more than half of the students submitted at least 12 hours before each assignment was due, with noticeable differences between test- and code-submission times in the upper-level course.

We also examined how long students had to wait to receive reviews on submitted work. The following table shows summary statistics on (a) the hours between artifact submission and receipt of *all* reviews, and (b) the hours before the receipt of the *first* review. These data show that students do get some feedback reasonably quickly. In future work, we will look at whether early feedback differs in review helpfulness or quality compared to later feedback.

		All Reviews (hrs to receipt)		**First Review** (hrs to receipt)	
Course	**Stage**	**Mean**	σ	**Mean**	σ
Upper	Tests	12.33	19.45	4.59	7.81
Upper	Code	11.54	20.9	4.66	11.89
Freshman	Tests	6.03	9.7	2.38	4.07
Freshman	Code	5.98	9.41	2.62	5.46

5.2 Student Reaction

Post-course, we surveyed students about the impact of reviewing on each individual assignment. We asked whether

[4]We deemed this assignment the hardest; students also rated it as benefitting the most from reviewing (see Section 5.2).

each of reviews received and reviews written were *not*, *somewhat*, or *very helpful* in improving their work. We also asked whether the review process was more helpful for *tests*, *code*, or *equally on both*. Each of the three questions was presented as a grid with assignment names labeling the rows and answer options labeling the columns. We received responses from 36 (of 49) students in the freshman course and 16 (of 37) students in the upper-level course.

The following table summarizes the survey results as percentages of students giving each response. The data from the freshman course reports on 3 of the 4 assignments. The omitted assignment was vastly easier than expected across the class, leaving students little to learn from the reviewing process (over 60% reported reviewing as not helpful on that assignment). The data from the upper-level course is broken into two groups of assignments: the programming assignments (reviews written on each of test cases and code), and the written assignments (one stage of reviewing only).

How helpful was (receiving/writing) reviews?				
Course	**Task**	**Not**	**Some**	**Very**
Freshman	Receiving	26%	59%	15%
Freshman	Writing	25%	45%	30%
Upper-Program	Receiving	45%	38%	16%
Upper-Program	Writing	43%	43%	14%
Upper-Written	Receiving	44%	40%	16%
Upper-Written	Writing	34%	28%	39%

The data on writing reviews on written assignments in the upper-level course particularly stands out ($\chi^2 = .05$ using the data on receiving reviews on writing assignments as expected values; this is significant at 97.5% with df=2). We hypothesize that this means that students felt they benefited from *seeing* each others' solutions, regardless of whether they had to write reviews on those solutions. Writing reviews is also rated "very" useful more often than receiving reviews within the freshman course. We are not sure how to interpret the difference in "not" ratings between the freshman and upper-level course: upper-level students may simply be more comfortable rating an aspect of the course negatively than students in their first semester; other hypotheses are also plausible.

On one assignment (in the freshman course), the "very" and "somewhat" percentages were identical on the helpfulness of writing reviews ("very" was lower in all other programming assignments in both courses). This was the only assignment in which we had students review not just tests and code, but also a *data definition*. The data definition was the first step of the problem: the students first defined a representation of a tree zipper, and then wrote various tree operations using the definition. Picking a good representation strongly guides an implementation towards a correct solution, and many representations cannot have implementations that are efficient (the assignment mandated big-O time bounds on the tree operations). In future work, we need to have students review data definitions on more problems, to help us assess whether the utility of reviewing varies across more than two problem stages.

In the future, we need to better understand the conditions that made the process more or less helpful. At mid-semester, students reported frustration trying to provide useful reviews on work that looked good overall: a student who did well might find the process unhelpful because the reviewers had offered little in the way of comment. Ratings of "not helpful" from students who submitted good work would mean something quite different than from students who submitted work with noticeable flaws. Deeper insight about the effectiveness of reviewing will come from analyzing the accuracy of students' reviews relative to the quality of the submitted work. We intend to provide a detailed analysis on these quantitative issues in future work.

On the question of whether reviewing is more useful on test cases or code, the two courses yielded opposite responses. The following table summarizes the percentages of students in each course choosing each option, averaged across the assignments for that course (omitting writing assignments).

Which of test or code reviews was more helpful?			
Course	**Tests**	**Code**	**Equally**
Freshman	24%	49%	28%
Upper	49%	27%	24%

All but one assignment in each course had subtleties that testing could expose. These results could simply reflect better appreciation of testing among upper-class students; future work should correlate these ratings with surveys of student attitudes towards testing in the vein of Buffardi and Edwards's work [1]. The upper-level course results, which emphasize the value of reviewing tests, particularly support CaptainTeach's multi-stage approach.

6. RELATED WORK

There is extensive literature on the benefits and pitfalls of peer-review in higher education and in computer science. We do not review the general literature here. Rather, we focus on research that touches on our foci: online reviewing, in-flow reviewing, multi-stage reviewing, or reviewing test cases. Topping's 1980–96 literature survey on peer review in higher education [11] predates uses of review in CS courses similar to in-flow and multi-stage reviewing.

Søndergaard [10] uses in-flow peer review in a compilers course. Students review after completing three compiler stages but before two others. Surveys show 68% of students agreeing that peer review helped improve their own work, 63% agreeing that it improved their ability to reflect on their own learning and skills, and 89% reporting value in seeing other groups' solutions.

Expertiza [7] allows for multiple rounds of revision and review, but on complete submissions, rather than on intermediate stages of assignments as in CaptainTeach. Expertiza also allows students to review one another's reviews, and these meta-reviews are used in assessing grades. CaptainTeach also allows students to give feedback on reviews, but we do not use the review process in assigning grades.

Hundhausen, et al. [5] use code reviews to help students develop soft skills in CS1. They study several variations such as on-line versus face-to-face, inclusion of a moderator, and re-submitting work after review. Their reviews do not consider testing. Their online process has students submit reviews individually, with an optional subsequent period for group discussion of reviews. They view the failure to require group discussion as more critical than the decision to conduct reviews online. The complaints made by our students are similar to ones they report.

Reily, et al. [8] study the accuracy, effectiveness, and impact of post-flow peer-review of programming assignments in an introductory Information Systems course. Their process requires submitting at least 3 (later 5) concrete test

cases as part of each review. Reviews also include Likert and open-response questions on various code characteristics. The sample review report in their paper suggests that their test cases were higher-level than ours, and at least some of them would be tested through manual interaction with a UI. In contrast, our test cases are for individual functions, and are expressed entirely in code. Their evaluation does not consider the role of testing in reviews, focusing instead on aggregation of reviews for accuracy and impact.

Zeller [13] presents the Praktomat system for submitting, automatically testing, and code-reviewing assignments. The reviewing is of whole assignments, with no in-flow component. In addition, only the final submission is assessed, rather than intermediate submissions. To combat plagiarism they personalize assignments, which results in a different reviewer experience from CaptainTeach as reviewers are longer reviewing the same problem they just solved.

Clark [2] presents a methodology where students are put in pairs or groups to review and test one another's solutions. This happens after an entire program has been created, but the projects run over a semester, giving this an in-flow feel. Students can use the outcomes of the review and peer tests to improve their solutions. The authors state that this caused students to work more steadily and consistently across the term of the long project.

Wang, et al. [12] use a workflow of early submission, followed by peer review, followed by resubmission, with the goal of getting students to do work earlier. They report that 88% (N=79) of their students "believed that their time management ability has been improved" after doing 10 assignments through this workflow. If we take the early test submissions in our data as an indication of good time management, this suggests similar results to our findings. While they have multiple steps of review possible before instructor review, students still submit the entire assignment at once, in contrast to the staged assignments of CaptainTeach.

Smith, et al. [9] use peer testing in a freshman honors data-structures course. After submitting a pair-programmed assignment, each pair tests code submitted by four other pairs. Students have three days to provide detailed reviews. Pairs then evaluate the reviews, summarize what they learned, and can submit updated solutions. To prevent copying, the course staff obfuscate and compile programs before distributing them. Our approach is much lighter-weight: it uses black-box tests primarily to diagnose whether students understand a problem and its corner cases, and expects students to spend much less time performing reviews. We would expect our students to learn less about testing per se than under Smith's approach, though we would expect them to gain a similar appreciation for good test cases.

Kulkarni, et al. [6] study how to scale peer-assessment to large online courses, where face-to-face reviewing is not feasible. To calibrate student reviews for accuracy, students first review one of a handful of assignments that have been assessed by TAs. Per assignment, students only review peers after writing a review whose score approximates that of the TAs. This approach takes training of reviewers more seriously than our approach of spot-checking reviews on seeded solutions, but the stakes are higher in Kulkarni's work as they use peer assessments for actual grades. Their rubrics are more detailed and structured, in part due to their more open-ended assignments. They also experiment with different formats to prompt for better reviews.

Buffardi and Edwards [1] study student engagement in *test-driven development* (TDD). We use tests first, but do not rigorously follow the details of TDD. Neither this paper nor Edwards' other research on testing uses peer review on tests. In a control-group-based study, they do not find significant differences in attitudes towards TDD in students who received feedback (hints) from automated test analysis, versus those who received generic hints not driven by tests.

Denning, et al. [3] explore lightweight in-class peer review. This provides students immediate, high-level feedback and helps instructors assess a class's understanding of a problem. In-class reviews are much lighter-weight than those in CaptainTeach, and done on problems that allow for such quick review. This work shares our goal of quick-turnaround reviewing while problems are fresh in students' minds.

Acknowledgments.

This work was partially funded by the US National Science Foundation and by Google. We thank the staff of Brown's CSCI 0190 and CSCI 1730 for their invaluable assistance, and the students for their forbearance.

7. REFERENCES

[1] K. Buffardi and S. H. Edwards. Impacts of adaptive feedback on teaching test-driven development. In *SIGCSE Technical Symposium on Computer Science Education*, 2013.

[2] N. Clark. Peer testing in software engineering projects. In *Australasian Computing Education Conference*, 2004.

[3] T. Denning, M. Kelly, D. Lindquist, R. Malani, W. Griswold, and B. Simon. Lightweight preliminary peer review: does in-class peer review make sense? In *SIGCSE Technical Symposium on Computer Science Education*, pages 266–270, 2007.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2002.

[5] C. D. Hundhausen, A. Agrawal, and P. Agarwal. Talking about code: Integrating pedagogical code reviews into early computing courses. *ACM Transactions on Computing Education*, 13(3), Aug. 2013.

[6] C. Kulkarni, K. P. Wei, H. Le, D. Chia, K. Papadopoulos, J. Cheng, D. Koller, and S. R. Klemmer. Peer and self assessment in massive online classes. *ACM Transactions on Computer-Human Interaction*, 2013.

[7] L. Ramachandran and E. F. Gehringer. Reusable learning objects through peer review: The Expertiza approach. In *Innovate: Journal of Online Education*, 2007.

[8] K. Reily, P. L. Finnerty, and L. Terveen. Two peers are better than one: Aggregating peer reviews for computing assignments is surprisingly accurate. In *Proceedings of the ACM International Conference on Supporting Group Work*, 2009.

[9] J. Smith, J. Tessler, E. Kramer, and C. Lin. Using peer review to teach software testing. In *International Computing Education Research Conference*, 2012.

[10] H. Søndergaard. Learning from and with peers: The different roles of student peer reviewing. In *ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 31–35, 2009.

[11] K. Topping. Peer assessment between students in colleges and universities. *Review of Educational Research*, 68(3):249–276, 1998.

[12] Y. Wang, H. Li, Y. Sun, J. Yu, and J. Yu. Learning outcomes of programming language courses based on peer code review model. In *International Conference on Computer Science & Education*, 2011.

[13] A. Zeller. Making students read and review code. In *ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2000.