Abstract of “In-flow Peer Review of Examples in Example-First Programming” by Joe Gibbs Politz, Ph.D., Brown University, May 2016.

Test-first development and peer review have been studied independently in computing courses, but their combination has not. In in-flow peer review, students provide feedback to one another on intermediate artifacts on their way to a final submission. We report on multiple studies of courses in which students conducted in-flow peer review of tests while assignments were in progress. We discuss student engagement, the correlation between review ratings and staff-assessed work quality, and evidence that test suites improved during the review process. Further, we present a useful distinction between full test suites and sweeps, which are small illustrative examples suitable for early submission and review. We show that these small sets of examples have reasonable quality, and are also a good target for peer review; for example, students suggest new tests to one another in a review over half the time.

In-flow Peer Review of Examples in Example-First Programming

by
Joe Gibbs Politz
B.S., WPI, 2009

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2016

This dissertation by Joe Gibbs Politz is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Shriram Krishnamurthi, Director

Recommended to the Graduate Council

Date _____ _____

Jeff Huang, Reader

Date _____ _____

(WPI Computer Science) Kathi Fisler, Reader

Date _____ _____

(UMass Computer Science) Arjun Guha, Reader

Approved by the Graduate Council

Date _____ _____

Dean of the Graduate School

Acknowledgements

To Mom and Dad, for the sprinting start.

To Caitlin, for all the other things in life.

To Chris, for consistency and confidence.

To Sarah, for commiserating.

To my thesis committee, for guidance.

And to all my teachers, especially:

Shriram, for feedback, navigation, and tenacity.

Kathi, for asking the right questions.

Mark, for the joy and craft of programming.

Dan, for clear and careful thinking.

Neil, for enthusiasm and initiative.

Mr. Townsend, for providing data and getting out of the way.

Ms. DiLeo, for enabling.

Ms. Bloom, for just doing out the darn math.

Mr. Boisselle, for Travelling Tales and touch typing.

Mr. Moran, for Big Dan and Little Ann.

Ms. Anderson, for surface tension and molecules.

Contents

List of Tables

List of Figures

Chapter 1

Thesis Statement

This document reports on the following claim:

> In-flow peer review of test cases can provide students with actionable feedback during programming assignments, and can help students improve their tests and their understanding of the problem.

Chapter 2

Background and Contributions

Peer code review has been employed for various reasons in Computer Science courses [26]. It is a mechanism for having students read each others' work, learn how to give feedback, and can help with assessment. The importance of peer review skills shows up in curricula at (at least) a national scale: of the six major computational thinking skills listed in the current draft of the AP Computer Science Principles curriculum [25], the fourth is:[1]

> P4: Analyzing problems and artifacts
>
> The results and artifacts of computation and the computational techniques and strategies that generate them can be understood both intrinsically for what they are as well as for what they produce. They can also be analyzed and evaluated by applying aesthetic, mathematical, pragmatic, and other criteria. Students in this course design and produce solutions, models, and artifacts, and they evaluate and analyze their own computational work as well as the computational work that others have produced.
>
> Students are expected to:
>
> - Evaluate a proposed solution to a problem;
> - Locate and correct errors;
> - Explain how an artifact functions; and
> - Justify appropriateness and correctness.

Peer review clearly has a role to play in developing each of these skills. Students are forced to read and evaluate proposed (partial) solutions, try to at least locate (if not offer corrections to) errors, offer their explanations for what an artifact is doing (especially if it does not match the expectations set by the problem), and justify their views on the appropriateness and correctness of presented solutions. Giving authors the ability to respond to reviews further reinforces the quoted principles.

[1] The CS Principles curriculum website is `http://lp.collegeboard.org/ap-computer-science-principles`

2.1 In-flow Peer Review

In-flow peer review (IFPR) [7] is a contributing student pedagogy [14] in which peer review happens before the assignment is due, on stages of an assignment that can be reviewed leading up to a final deliverable. One of the main goals of IFPR is to motivate students to engage with peer feedback by giving students a chance to respond to it while it is still relevant for the assignment (and their grade).

The design of an in-flow assignment centers on choosing the intermediate submissions for review. Intermediate submissions could be complete drafts, or components (like individual functions) that build up to the final product. These submissions should be manageable for students to review relatively quickly to encourage prompt and actionable feedback. In addition, since IFPR occurs while the assignment is out, submissions can also be useful to the reviewers themselves, who benefit from seeing examples of others' work in the middle of their assignment.

2.2 The Design Recipe

Consider this data-structure programming problem:

> 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, 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 (HTPD) [12] 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 example uses of 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 the second step).
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 examples against the function, and iteratively add more tests and improve the body of the function until it is complete.

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 HTDP 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 until the prior steps are done). In our experience, 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.

This step-by-step process is a natural fit for in-flow peer review. Each step of the design recipe is a moment for reflection and assessment, and as such they make excellent opportunities for peer feedback. In principle, we could build programming environments that walk students through this workflow for problems, and integrate review at the most valuable steps. That is a larger vision than this document addresses. Here, we report on first steps towards integrating the design recipe with in-flow peer review.

2.3 Example-first Programming

In this work, we choose to focus on the example-writing step—step 4—of the design recipe as a target for peer review. Writing examples first has a number of benefits for students on its own, before considering review. It forces programmers to think through the problem separate from the implementation. The examples can be the start of a test suite that helps debug a partial solution while it is being written. Finally, a set of examples written first encourages testing of the problem itself, not simply the particular implementation that was chosen; this is relevant when implementing a program that may have more than one right answer for a given input.

In general, test-first development strategies involve writing tests before and alongside an implementation. Depending on the complexity of the program being written, it might not be feasible to write all the interesting tests first. As such, we don't take the rigid stance of test-driven development, in which tests must be written and observed to fail before implementation begins, in our definition of test-first programming. We encourage students to write examples early and without an implementation when possible, but do not require exhaustive testing up front. We distinguish these two cases terminologically, using *examples* to refer to a representative exploratory set of checks, and *tests* or *test suites* to refer to a set of tests intended to find bugs and gain in implementations.

2.4 Review of Examples

The same reasoning that makes examples attractive as a tool for TAs to help students with their work also makes examples a potentially useful artifact for review. There are other benefits to examples as a review target, as well:

- They are separate from the solution program itself, so issues of plagiarism that could result from submitting entire program drafts are ameliorated.

- Judging their correctness is a concrete activity – they either produce the correct outputs for given input or they don't.
- Judging their comprehensiveness is more difficult, but the feedback can be made actionable – a reviewer can give a description of missing cases or sparsely-tested parts of the input space.
- They can be useful for reviewers, who have the chance to see cases they may have missed or not considered, and compare their understanding of the problem to others'.
- They can be assigned at reasonable scale for review (we discuss tradeoffs of submission size more in section 6.1), of a few hundred lines at the absolute limit. This amount of review is manageable in a class assignment setting when assignments are on the scale of a week or two. This limit is both practical and motivated by, e.g. a large and recent industrial case study [8] that suggests: "the single best piece of advice we can give is to review between 100 and 300 lines of code at a time and spend 30-60 minutes to review it."

2.5 Contributions

The rest of this document argues for different facets of the utility of review of examples, and necessary support for example-first review workflows. There are several concrete research questions and objectives for this work.

- Does review of tests in test-first programming help students improve their test suites?
- Does review of tests in test-first programming help students improve their programs?
- What is the relative usefulness of receiving review feedback on tests versus seeing other students' test suites and composing feedback?
- What are example assignment workflows, testing strategies, and types of tool support (e.g. submission and review assignment systems) that enable effective reviews of tests?
- What software support is necessary to deploy in-flow peer review of examples in classrooms?

2.6 Non-Goals

In our analysis, we are explicitly studying the effectiveness of *review of tests, focusing on examples*, not review in general or example-first programming in general. It is not the purpose of this study to determine if review of tests is more or less effective than review of code, for example, or to compare example-first to test-last programming or test-driven development, though this is discussed in chapter 9. These studies do generate insight into useful comparative questions to ask about other kinds of review or programming strategies (section 8.2), but they are not the focus.

Chapter 3

The Design Space of Example and Test Review

In order to effectively set up an assignment for peer review of examples, it needs to have (or be altered to have) a few specific qualities. Obviously, it first needs to define what it means to test the programming problem at hand. Since students need to see one anothers' work during review, the workflow needs to enable submission of the initial examples stage, and the assessment needs to take plagiarism issues into account. Finally, the assignment workflow needs to provide for assigning reviews between students, and distributing the feedback that students submit.

This section lays out some interesting facets of the design space for assignments that use test-first review. In chapter 4, we discuss how the assignments we use in studies address each of these points.

3.1 Requirements on Programming Problems

The programming problem needs to have a testable specification that is independent of a particular implementation; for example, input/output pairs, pre- and post- conditions, or a rubric for output quality (for, say, graphical output).

One way to do this is to fix the specification of data for the assignment and use a functional specification for the implementation. This leads to test suites of independent input/output pairs, which may be particularly amenable to review, since their behavior cannot interact and they can be reviewed in isolation.

As an example in this section, consider an assignment with the following specification:

> *Implement a function, `sort`, that takes a list of Person data structures, and returns a new list, containing the original Person data structures sorted lexicographically by name.*

A potential code and test setup for this example is shown in Figure 3.1, implemented in the Pyret programming language.[1] An example or test is a call to `sort` (given a list of `Person` structures), paired with an

[1] `http://pyret.org/`

```
data Person:
  | person(name :: String, age :: Number)
end

fun sort(people :: List<Person>) -> List<Person>:
  # implementation goes here
end

check:
  # A sample test, which defines two people and checks
  # that the returned list is sorted
  darryl = person("darryl", 28)
  carol = person("carol", 40)
  sort([list: darryl, carol]) is [list: carol, darryl]
end
```

Figure 3.1: A sample test-first setup

expected output list. Students may or may not be given this example at the outset, depending how familiar they are with testing so far in the course.

The check block is clearly a separate artifact from the implementation, and can be submitted, evaluated, and reviewed on its own. Multiple students' check blocks can be compared, and what a student learns from writing their own can be applied to understanding other students'.

The same reasoning applies to unit tests in a framework like JUnit or PyUnit that put tests in a separate file, or even external shell scripts that drive a program with a command-line interface. The tests just need to be a separately-specified artifact from the implementation.

3.2 Submission and Review Logistics

There are several design decisions and concerns relative to assigning and performing reviews.

Separate Submission The assignment submission system needs to handle submission of examples (in the last section's example, the `check` block), separately from the implementation, and before the implementation itself. This could be via a separate file, via copy-pasting the check block itself into a web-based submission system, or a more sophisticated system that tracks a student's editing progress.

Intermediate and Review Deadlines There is a decision in whether there is a separate *deadline* for tests, in addition to a separate submission step. If students are sufficiently motivated by review, it may not be necessary to force early submission, and instead simply have test submission as a separate step that must be completed before review. However, to enforce that all students get reviews in time to use the feedback, it could be useful to enforce a test submission deadline several days before the assignment is due.

Review Assignment Timing With the exception of explicitly assigned review groups (discussed in chapter 5), our workflows always assigned submissions to reviewers as soon as possible. In some cases, this could mean that two students acted mutually as reviewer and reviewee, which was the case with the review assignment discussed in section 6.1. With instructor-provided review artifacts provided for the first submitters, as was done in chapter 4, there is no mutual overlap. Either way, we elected to not wait for a checkpoint in the assignment to do reviews, and instead elected to get students started giving feedback as soon as possible. This is just one possible choice we could have made, and there are tradeoffs; some alternatives are discussed in section 8.2.

Anonymity Another choice we had to make was whether to show the identity of reviewer and/or reviewee during the review process. There are some good arguments for showing identity: it reflects real-world review practices, where developers give each other candid feedback tagged with their names; it allows students to continue conversations outside the review system; and it allows students to meet and interact with other students which could be useful for choosing partners in group work. Despite these potential benefits, we elected to make the reviews in our courses anonymous.

We made this choice for a few reasons. We didn't want students to feel intimidated by either their reviewer or reviewee, which could happen with a (perceived) weaker student paired with a (perceived) stronger student. We hoped it would make students more confident in their reviews and submissions, since they didn't have to face personal embarrassment if they ended up being wrong. We were monitoring the system for overly negative or harsh reviews, so we didn't feel that we needed identities shared for any kind of accountability. Different courses may feel the need to share identities for the positive reasons outlined above; we erred on the side of caution and didn't share this information.

3.2.1 Plagiarism

One concern in a setting where students see one another's code by design is the potential for copying solutions. In all of our in-flow review assignments the tests step that is reviewed is resubmitted in some form at the end of the assignment, and part of the goal is for students to improve their tests by the end. Since they see others' code along the way, we need to address what happens if they copy tests into their own test suite for final submission.

First, we note that students *do not know* that the examples they are viewing during a review are correct. If they blindly copy examples that they see in review, they could actually degrade the correctness of their test suite (a metric we explicitly measure). We will also see in section 4.1 that in one setting, we seeded the system with intentionally incorrect submissions so students could be assured they needed to exercise judgement. So blind copying, while it may be productive on average, is far from a perfect strategy.

Second, we discuss in section 6.2 and section 4.3 how we weight initial and final submissions separately in grades. This discourages the strategy of submitting an empty initial set of tests and just using what's copied during review. A student who does this would get no credit for their initial submission, which is given enough weight that it would be detrimental to their grade to skip it. At the same time, their final submission is given significant weight as well, so there is motivation to improve over the course of the assignment.

3.3 Motivating Examples, Testing, and Review

Students may be interesting in participating in test review simply out of enlightened self-interest combined with altruism towards their classmates. However, some class settings require extrinsic motivation (like an effect on students' grades), to ensure that students will participate in early submission and review. In addition, students trying to game the system in a course setting with ungraded tests might intentionally submit poor tests to avoid giving their classmates any hints about the problem.

Both the initial test submission and the reviews could be graded in order to motivate students to both submit good tests and take review seriously; grades are a straightforward kind of motivation to give students in traditional classes that are assigning grades to student work already. Grading reviews themselves can take a lot of instructor and TA effort, since there can be many more of them than actual submissions, but there is no issue in theory with making review quality part of students' grades.

In the courses in this study, we have chosen to grade initial test submissions, but *not* grade reviews themselves. We read reviews to get a measure of their quality, but we have the instructor simply send email to students when we are disappointed in their reviewing quality, without attaching a grade to it. We hope to motivate students to participate merely out of a desire to help their classmates and participate in a collaborative process that improves everyone's work.

3.4 Evaluating Examples and Tests

When examples and testing are an explicit part of an assignment, we need a mechanism for assessment. There are a number of metrics used to grade examples and tests.

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 tests could be held in reserve until the end of the assignment and assessed then. But it is still difficult for a student to predict whether a particular example would cover all paths in a final implementation since that is dependent on a yet-unknown structure.

Another technique sometimes used is running students' tests against one another's implementations, and giving points for bugs caught [10]. Again, this requires waiting until the end of the assignment to assess the examples.

In order to get a metric for test suite assessment, let's revisit two metrics for evaluating tests earlier:

- *Correctness* – do the tests 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 by tests, not one particular implementation. For example, a function that involves a division might have tests with perfect code coverage that are nonetheless lacking 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 tests. 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* [9], an existing technique in software engineering for ensuring the quality of a testing process.

Specifying assignments so that they are separately testable is challenging. For some assignments, we assign an examples step and review, but use TAs to grade tests more holistically and relative to the student's implementation. This decision is often made on an assignment-by-assignment basis, because relatively small omissions in assignment specifications can lead to students submitting interface-violating test suites. For example, if the assignment description neglects to mention that students shouldn't test the order of output lists that represent sets, it becomes much harder to automatically gauge the correctness of their tests.

Better mechanized support for checking that students match a particular interface *while the assignment is in progress* is thus complementary to review of examples, as it helps avoid the case where students submit work that needs to be graded by hand.

Chapter 4

Test Review and In-flow Peer Review

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, which we call CS1.5 in this document,[1] the other an upper-level programming languages course, which we call CSPL in this document.[2]) We built an enhanced programming environment, Captain Teach 1.0 (CT1), to support the reviewing process. For the programming assignments in this section, CT1 used the programming language Pyret, whose design supports multi-stage reviewing (chapter 7 discusses how). This section describes the assignment structure in those classes, how CT1 was used for review, the data that was collected, and the results of analysis on that data. This chapter contains material covered in two publications by Politz, et al. [18, 19].

4.1 Review Setup

Review Assignment Students performed review on both tests and implementations in this course. They were assigned submissions to review immediately upon submission of a test or implementation step. Students were assigned a mix of student submissions and TA-provided solutions, based on the following criteria:

- Half the time, they were assigned N other student reviews. These reviews were chosen as the oldest submissions from those with the least reviews (so a submission with no reviews would be chosen over one with a single review, but among two submissions with a single review, the older submission would be chosen). N is a parameter of the assignment, either 2 or 3.
- The other half of the time, they were assigned (N-1) student reviews according to the same process as above. They were also assigned 1 TA-provided solution, which would be *known-good* half the time, and *known-bad* half the time.

To accommodate the first N students to submit, the system was also seeded with N staff-provided solutions, designed to look like reasonable student solutions (some contained intentional minor mistakes). These

[1] `http://cs.brown.edu/courses/cs019/2013/`

[2] `http://cs.brown.edu/courses/cs173/2013/`

were treated like normal student solutions, and could be assigned reviews multiple times.

The *known-good* and *known-bad* solutions were used to give students immediate feedback on if they had reviewed badly, described in the next section. Known bad solutions were typically designed to be notably bad rather than to trick students, as the goal was to ensure that students got some amount of negative feedback if they simply clicked through, always giving positive or negative reviews.

4.2 Review Rubrics

When presented with a test review, students were shown the following two prompts, inserted inline into the editor viewing the tests under review. Each question was followed by a 6-point Likert scale from "Strongly Disagree" to "Strongly Agree" and a text box for open-response comments.

- "These tests correctly reflect the desired behavior."
- "These tests are representative of the possible inputs."

The goal with the review prompts was to get at a concrete *correctness* question, and also asking about the *thoroughness* of the test suite.

For implementation reviews, they were asked the following two questions, followed by the same Likert and free response feedback form.

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

Again the first asked a correctness question. The second question focuses on the design of the implementation. We don't discuss the structure of the code rubric further here, due to our focus on test review.

Students received email each time a review of their work was completed by a classmate. They could see all of their reviews through the same Web-based interface they used to author their submissions.

Students received two kinds of feedback on reviews themselves:

- The system would give an immediate popup for reviews of the *known-good* and *known-bad* solutions. If a student indicated that the tests were correct on a known-bad test suite, or incorrect on a known-good one, the popup would indicate the mistake. Similarly, if the student was correct, it would give them a popup telling them so. We hoped that this would prompt students to not blindly sumbit all-correct or all-incorrect reviews, and engage with the system more.
- For reviews of students' work, students could (optionally) give the reviewer feedback on the helpfulness of the review. This was presented as a Likert scale of helpfulness, along with a free-text area, along with the review.

Finally, the students could also *flag* a review to bring it to the attention of the course staff if it was deemed inappropriate or offensive. No one used the ability to flag a review in either course.

4.3 Grading and Policies

The grading methodology differed between the two courses.

In the upper-level programming languages course (CSPL), students tests were graded automatically based on the gold/coal strategy described in section 3.4. Their implementations were graded by running them against a test suite written by the TAs. Their final grade was a weighted combination of four components: initial/final test score, and initial/final code score. Heavier weight was given to the initial submission, which varied per assignment from 60-75% of the total. The split between tests and implementation also varied, though it was closer to an even split.

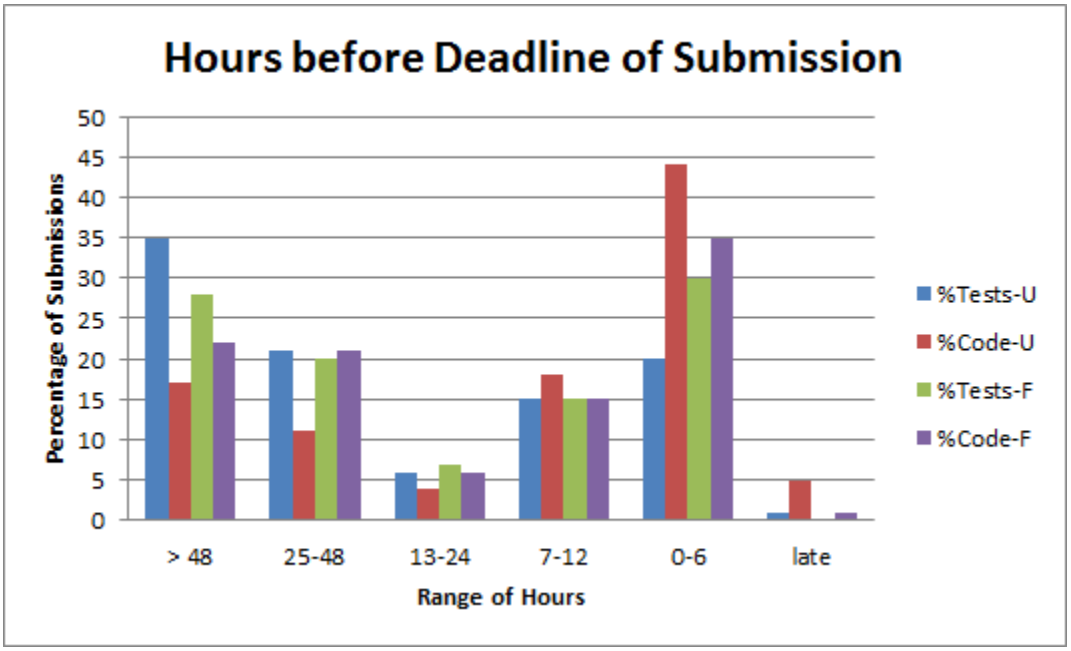
In the lower-level introductory course, grades were much more heavily based on TA judgment on style and comprehenseiveness of testing. These assignments had more individual pieces and were more difficult to design coal solutions for. Most implementations were still graded automatically against a TA-written test suite. TAs graded initial submissions first, then took any alterations in the final submission into account to improve students' grade.

4.4 Data Collected

Running the course resulted in data in the form of student submissions and reviews, as well as student interactions with the system. In addition, at the conclusion of the semester, we surveyed the class on their experiences with review. This section summarizes the data that was available for analysis at the end of the semester.[3]

- **Survey Data** For each assignment, we asked students whether *receiving* or *writing* reviews was more helpful, and whether *test* or *code* reviews were more helpful. 36 of 49 students in CSPL and 16 of 37 in CS1.5 responded.
- **Submission Data** The contents of each intial and final submission, along with the timestamps of when they were submitted, for each student and each assignment.
- **Review Data** The Likert and free-response review data for each student and each review, the time the review was submitted, and the time the review was viewed (if ever) by the reviewee.
- **Feedback Data** The Likert helpfulness score and free-response helpfulness feedback, and the time it was submitted and viewed (if ever).
- **Assessment Information** The golds and coals, and reference test suites, used to evaluate students tests and code.

[3]For readers interested in the IRB status of this work: Since the students' experience wasn't altered to fit the study, as the peer review tool was going to be used in class anyway, and the data was anonymized for analysis, we determined that this did not qualify as human subjects research. This holds for the work in future chapters, as well.



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.

Figure 4.1: Submission timing for CSPL and CS1.5 in 2013

4.5 Analysis and Results

There are several analyses we performed on the data from the 2013 courses. This section includes results published in two publications by Politz, et al. [18, 19].

4.5.1 Submission Timing

Figure 4.1 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

		All Reviews		First Review	
Course	**Stage**	**Mean**	σ	**Mean**	σ
Upper	Tests	12.33	19.45	4.59	7.81
Upper	Code	11.54	20.90	4.66	11.89
Freshman	Tests	6.03	9.70	2.38	4.07
Freshman	Code	5.98	9.41	2.62	5.46

Table 4.1: Time from submission to first and all reviews (times given in hours)

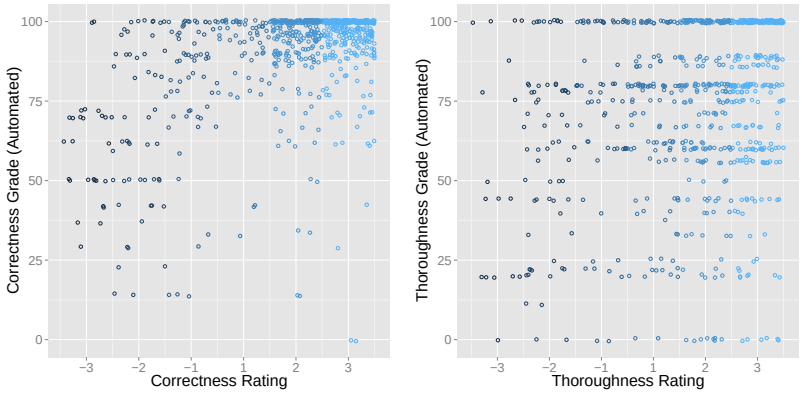


Figure 4.2: Review ratings versus grades on correctness and thoroughness

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 table in Table 4.1 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.

4.5.2 Review Accuracy

For reviews to be an effective form of feedback, they should be accurate. To check whether students are accurate reviewers, we compare their ratings to the gold/coal grades. Figure 4.2 plots the Likert review ratings against the gold (correctness) and coal (thoroughness) grades in CSPL. The plots show that initial submissions are generally strong, and that positive reviews outweigh negative ones. Thoroughness reviews are less concentrated in the upper-right quadrant than correctness ones. The horizontal bands in the thoroughness plot reflect the nature of the grading: thoroughness grades are a percentage of a single-digit number of coal solutions, so only a few grades are possible per assignment.

The interesting sections of these graphs are the lower-right and upper-left portions, which reflect reviews that were inconsistent with our grading assessment. Possible explanations in these cases include poor performance of the reviewer on that assignment, last-minute reviewing, weaknesses in our grading mechanisms (that failed to accurately assess work quality), or lack of engagement by the reviewer. For each of thoroughness and correctness, we manually inspected all data points with the top two Likert ratings and grades below 40, as well as points with the bottom two Likert ratings and grades above 60.

For the 43 cases of low thoroughness reviews with high grades, all but a couple of the free-form comments point to concrete situations or constructs that the test suite did not adequately exercise. This strongly suggests that this segment of the plot gets populated due to limitations in our suite of coal solutions.

For the 45 cases of high thoroughness reviews with low grades, just over half gave generic comments

	both	gained	lost	missed	total
# instances	1252	155	5	284	1754
seen as reviewer	1209	138	4	183	1590
reviewer caught it	1164	129	5	166	1516
not seen	11	5	0	47	69

Table 4.2: Coals caught before and after review

of the form "this is great", while a dozen stated specific criticisms and potential holes in the test suites that would be consistent with lower ratings. Those dozen might reflect review inflation. For the others, we looked at reviewer performance and review submission time, but found no patterns.

The comments for the 46 reviews with low correctness ratings but high grades are also detailed and concrete. These high grades could also be indicative of weaknesses in our tests. There are only 7 high-rated correctness reviews with low grades; the comments vary widely in style, and the sample is too small to draw inferences.

4.5.3 Observed Actionable Reviews

One useful source of feedback on the effectiveness of reviews was reviewee open response feedback, which was optional, but provided in 409 of 2428 reviews.

We manually coded all the review comments for indications that reviews had identified problems or led to edits in test suites (these were two of 15 categories identified during open coding of the comments). In the following table, "Ack Error" reflects feedback that acknowledged an error that was pointed out in a review, and "Will Act" captures claims that the reviewee would edit their test suite as a result of the review.

	Ack Error	Will Act	Both
# comments (CSPL)	40	27	2
# authors (CSPL)	20	13	2
# comments (CS1.5)	55	30	13
# authors (CS1.5)	25	15	10

While in absolute numbers these were small, the feedback was completely optional, so this indicates that in both classes, roughly half the students got a review that they took the time to indicate pointed out a problem or suggested a change that they would make with no requirement to do so. Assuming the students didn't give misleading feedback, it is safe to take this both as evidence that the review process is providing useful feedback to some students, and that these numbers are a lower bound on how often it is helping. A more detailed coding of review feedback on a different data set is discussed in section 6.3.7.

4.5.4 Test Suite Improvement

Recall that automated test grading runs student tests against so-called *coal* solutions with intentional errors. The more comprehensive the test suite, the more coal solutions will be caught, and a weak test suite will miss

some coals. Since students submitted both an initial and a final test suite, we can compare coals that were caught in initial versus final tests.

Table 4.2 shows the results of one analysis related to coal changes. The first row counts, across all coal-student pairs and across all assignments, how many coals were caught in **both** the student's initial and final submission, **gained** by not being caught initially but being caught by the final submission, **lost** by being caught initially and not caught in the final submission, and **missed**, by not being caught in either the initial or final.

The **gained** and **missed** columns are the most interesting to consider related to the review process, as we can ask under what conditions students were more likely to improve their test suite by gaining a coal. The rows break these numbers down relative to reviewing activity:

- The second row shows, of the totals in the first row, how often each case happened when the student *reviewed a test suite that caught the coal*. That is, 138 of the 155 times a coal was gained, there was some test in the suite the gaining student reviewed that they could copy-paste into their own suite to catch the coal solution. Compare this to 183 out of 284 times that it was not added before the final submission.

- The third row shows, of the totals in the first row, how often each case happened when the student *was reviewed by a student whose initial test suite caught the coal*. That is, 129 of the 155 times a coal was gained, the gaining student's reviewer wrote a test that covered a case they did not. Compare this to 166 out of 284 times that it was not added for the final submission.

- The final row shows how often, of the totals in the top row, they neither reviewed nor were reviewed by a student who caught the coal. That is, 5 out of 155 times a student gained a coal, the gaining student had no reviewing contact with another student's solution that caught the goal they gained. Compare this to 47 out of 284 times that it was not added before the final submission.

Continuing to focusing on the 439 instances in which a coal was not detected initially (the **gained** and **missed** columns), we find that 89% of those who gained the coal reviewed work that caught the coal compared to 64% who did not gain the coal. This difference is significant (p=.000065, χ^2=16.02), and suggests that reviewing impacts test-suite quality. Similarly, 84% who gained a coal were reviewed by someone who had caught the coal, compared to 61% of those who did not gain the coal (p=.00049, χ^2=12.14). This also suggests that being reviewed impacts test-suite quality. Note that because of the significant overlap between these two groups, it is difficult to tell if the difference is more strongly correlated with the receiving or giving end of reviewing from this data.

Note that there are a number of possible hypotheses that fit this data: It could be that the strong reviewers all tend to catch most of the coals, and being reviewed by a strong reviewer is more helpful than being reviewed by a weak reviewer. The same could be true for the most readable and accessible tests to learn from also tend to catch the most coals. Also, code review also happened before final test submission, and this analysis does not take any effects of that review into account. Thus, this result is merely correlational, and suggests that we should further investigate what parts of the review process may be involved in the observed difference.

Responses to: *How helpful was (receiving/writing) reviews?*

Course	Task	Not	Some	Very
Freshman-Program	Receiving	26%	59%	15%
Freshman-Program	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%

Table 4.3: Student survey responses on peer review effectiveness

4.5.5 Student Reaction

Post-course, we surveyed students about the impact of reviewing on each individual assignment. We asked whether 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 surveys themselves are available in "Appendix B – Review Surveys" on page 72.

The table in Table 4.3 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).

The data on writing reviews on written assignments in the upper-level course particularly stands out (χ^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.

Chapter 5

Review Groups

In 2014, peer test review was used in the same two courses as in 2013, and several others as well. We focus on the repeat courses and on details of review assignment in this section. In the next section, section 6.1, we examine a different review artifact in more detail. The assignment structure and software were redesigned based on the feedback and analysis from the 2013 courses.

5.1 Assignment Setup

The assignment setup changed over the course of the semester in response to both student feedback and planned procedural changes. In all cases, and in contrast to 2013, there was always only a single step of review on tests, with no review of implementations, and only a single set of instructions for testing even when multiple functions were being implemented. Tables 5.1 and 5.2 show the assignments and which review and test strategy was used on each. We outline the meaning of the labels in this section.

The plan for the assignments in both courses was to have an an initial assignment or two to get students used to the review system, where all students would participate in review equally. This corresponds to the assignments labelled **All**.

After these initial assignments, and as assignments ramped up in difficulty and complexity, the students

Assignment	**Who Reviewed?**	**Test Type**	**Rubric**
Oracle	All	Sweep	Normal
Tree Updaters	Groups	Sweep	SweepNoStructure
DocDiff	Groups	Sweep	SweepNoStructure
Brown Heaps	Groups	Sweep	SweepCount
Parallelism	Opt-out	Sweep	SweepCount
Tour Guide	Opt-out	Sweep	SweepCount
Fluid Images	Opt-out	Sweep	SweepCount

Table 5.1: CS1.5 Assignment Types

Assignment	Who Reviewed?	Test Type	Rubric
Basic Interpreter	All	Full	Normal
Records Interpreter	All	Full	Normal
Type Checker	Groups	Full	Normal
State Interpreter	Groups	Sweep	Normal
Java Interpreter	Groups	Sweep	SweepNoStructure
CPS Transform	Opt-out	Sweep	SweepNoStructure
Lazy Interpreter	Opt-out	Sweep	SweepCount
Type Inference	Opt-out	Sweep	SweepCount

Table 5.2: CSPL Assignment Types

were randomly split into three groups for each remaining assignment. One group would only *receive reviews*, one would only *write reviews*, and one would do *no reviews*. The group memberships would change from assignment to assignment to spread the reviewing workload. This review assignment strategy was used in the assignments labelled **Groups**.

The motivation for review groups was to reduce the overall review workload, which students reported (anecdotally) as being high. We expected the review groups had the potential reduce workload while still conveying the concepts we cared about in review.

In both courses, after a few assignments using the review groups, multiple students complained that they thought the groups were unfair. In response, a survey was issued (section 5.3) asking if they would prefer to always participate in review. The majority of students, in both courses, wanted to participate in review on all assignments, though a minority indicated that they would like to not participate at all. Motivated by these results, for the remaining assignments we did review on an opt-in basis, where students could choose, at the time of test submission, whether they want to participate in review or not. These assignments correspond to **Opt-out**.

5.2 Impact of Review Groups

Naturally, we were curious if participation in review groups affected students' performance. The most obvious hypothesis is that submissions made by students who didn't participate in review would catch fewer coals and have buggier implementations For the three CSPL assignments that used review groups we examined three different criteria: the number of coals caught in *final* submission, the number of incorrect tests in the final submission, and the number of reference tests passed by the final implementation.

Table 5.3 summarizes this information. Means are reported grouped by the three groups: **N**o review, **D**oes review, and **G**ets review. The f and p parameters for the one-way ANOVA are reported for each. Note that for GoldFail, higher numbers indicate a worse test suite, because it counts each *incorrect* student test. The other two assessments, CoalCatch and ImplTest, are both higher-is-better scores. Most were far from showing any statistically significant result that their means differed. The mean scores are sometimes better and sometimes worse for each group; there is no clear winner.

Assignment	No Review	Did Review	Gets Review	*f*	*p*
InterpState-CoalCatch	8.9	10.0	10.5	1.34	0.269
InterpState-GoldFail	0.9	0.2	0.5	1.37	0.262
InterpState-ImplTest	109.5	124.2	122.6	3.29	**0.044**
TypeCheck-CoalCatch	9.1	9.9	8.7	2.93	**0.060**
TypeCheck-GoldFail	3.3	3.3	4.1	0.31	0.734
TypeCheck-ImplTest	44.8	46.3	46.9	0.73	0.486
Java-CoalCatch	8.8	8.5	9.0	0.13	0.876
Java-GoldFail	2.1	2.5	1.5	0.96	0.388
Java-ImplTest	33.7	35.8	34.6	0.26	0.773

Table 5.3: Performance of review groups

One case does show a statistically significant difference, that of the implementation for the State Interpreter (the bolded 0.044). There are a few outlier submissions that are essentially incomplete and barely working for this assignment, and all three were in the no reviews group. These appear to pull the mean down enough to show the significant difference. Another case is nearly significant, the Type Checker CoalCatch (bolded 0.060), though interestingly the gets review group has the lowest mean in that case.

On these assignments, we conclude participating in review didn't significantly change performance by the main metrics used for assessing them.

5.3 Student Opinion of Review Groups

After students reported wishing they could participate in more review (and after the advent of the sweep), we surveyed students to find if they would prefer to always participate in review over the review groups setup. The results of the survey are in figure Table 5.4.

Both courses were asked a similar series of four questions, asking if they preferred to (1) participate in review in some form all the time, (2) participate in both writing and receiving reviews, (3) just write reviews (which includes seeing others' sweeps), and (4) just receive reviews. In CSPL, where the sweep was introduced after several assignments with full test suite submission for the review step, we asked an additional question about preference for the sweep style over full review style (CSPL-SweepSwitch). The full surveys are shown in "Appendix B – Review Surveys" on page 72.

The response showed that the majority of students preferred to participate in review all the time. In CSPL, the majority found the sweep helpful, though perhaps somewhat less enthusiastically, with fewer Strongly Agrees. There was also a significant minority of students who were lukewarm or outright opposed to review in general. This motivated two decisions:

- We would stop using review groups, and all students participating in review would participate as both reviewers and reviewees.

Question	RD	D	SD	SA	A	RA
CSPL-SweepSwitch	1	1	3	10	17	5
CSPL-SomeReview	1	3	3	2	15	13
CSPL-BothReview	3	2	1	6	11	14
CSPL-JustWrite	9	11	10	5	1	1
CSPL-JustReceive	7	12	7	2	5	4
CS1.5-SomeReview	0	1	0	8	12	20
CS1.5-BothReview	0	3	2	8	9	17
CS1.5-JustWrite	0	7	12	12	3	1
CS1.5-JustReceive	0	5	9	9	6	7

RD=Strongly Disagree, D=Disagree, SD=Slightly Disagree
SA=Slightly Agree, A=Agree, RA=Strongly Agree

Table 5.4: Responses to review survey during review groups period.

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 6.2 and Figure 6.1. Assignments 1 and 2 in CSPL did not give students a choice on review participation.

Table 5.5: Student choice in assignments with opt-out

- We would allow students to choose whether they would participate in review or not, at submission time, on each assignment. That way, we would avoid forcing students into reviewing, which we reasoned could lead to cursory reviews from unwilling reviewers. This decision was a key place where configurability of software (chapter 7) was crucial, since we could adapt the review workflow mid-semester.

5.4 Opt-out Assignment Participation

This section reports on work done in collaboration with Joseph M. Collard, Arjun Guha, Shriram Krishnamurthi, and Kathi Fisler [17].

For the last 3 assignments in CSPL and the last 4 in CS1.5, 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 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 Table 5.5. Between 30 and 40% of students did not see enough value in review of sweeps to continue doing it, 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. Review improvement from opt-out isn't inevitable, as it could be that the students who opt out are actually the best reviewers, so removing them from the review pool actually reduces the overall quality of review. For example, it's worth noting that CSPL has more advanced students and a more uniform cross-section of that department's population, while CS1.5 mostly consists of students who had significant high-school computing and all of whom placed into the accelerated course, and are thus more likely to view themselves as "hot shots". Student attitudes like these could affect which part of the student population participates. As the focus of our study is the sweep rather than peer-review configurations per se, we leave that question for a future study.

Chapter 6

The Sweep

This chapter reports on work done in collaboration with Joseph M. Collard, Arjun Guha, Shriram Krishnamurthi, and Kathi Fisler [17].

There was another major change we made in the courses we taught in 2014 not covered in chapter 5.

6.1 Suites vs The Sweep

From conversations with students (and from numerous complaints filed with TAs), we became aware that the test review was missing not quite hitting the mark of the exploratory *examples* step we intended students to be completing. Instead, we were asking them to submit a full set of tests, implicitly via the assessment of tests with gold and coal solutions, and explicitly with the way we worded assignments. There were three main reasons that this was a problem:

- 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 3.4) to write comprehensive test suites. This resulted in suites with hundreds of tests, which were too onerous to review. This resulted in poor-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

[1] "Sweep" is a term from archaeology for a segment of a dig site that can be surveyed for an non-comprehensive, but interesting, selection of artifacts (`https://en.wikipedia.org/wiki/Archaeological_field_survey#Fieldwalking_.28transects.`

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. Documentation for an API often comes with a small number of carefully-chosen examples. Exploring a new library for the first time often involves coming up with interesting examples inputs to try. Separating feature-critical, first-priority tests from more in-depth system tests is also a useful software engineering skill. Thus, the idea behind the sweep shows up in many later, professional contexts, even if not by that name.

A Google programming contest[2] provides a description of sample inputs and outputs that is apt, especially with regard to carefully chosen examples circumventing other potential communication issues like language barriers (emphasis added):

> Sample I/O isn't just here so your contestants can test their code: it's here so your contestants can **test their understanding of the problem**. If there's a special case in your problem's specification, put it in the sample input. Sometimes it's hard to describe a problem; if someone who doesn't read your contest's language very well might have difficulty understanding what to do in a certain case, put it in the sample input.

Of course, after producing a sweep, students must still eventually—with the final submission—construct a strong test suite, going beyond the initial examples. The sweep is a mechanism for starting to think and communicate about the problem, not a final solution. Shifting the focus of peer review from test suites to sweeps (examples) 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 and the data we collected for this study (section 6.3.1). We then examine the behavior of student sweeps and suites for thoroughness and correctness (section 6.3.2). Finally, we explore the students' reviewing activity when giving feedback on one another's sweeps (section 6.3.3).

29). We have also heard the term used in geology, to refer to a sample of rocks that give an overview of a geological site.

[2] `https://code.google.com/codejam/problem-preparation.html#sample`

6.2 Assessing Suites and Sweeps

The gold/coal technique for reviewing tests (section 3.4) applies well to both initial sweeps and to final suites, with one minor caveat. When we introduced the sweep, students (rightfully) pointed out that it would be unfair to grade sweeps against the same standard of coals as full test suites. To assess sweeps, we set a threshold of coals that needed to be passed for full credit that was some fraction of the total number of coals we used for grading sweeps in final submissions. Students are then assessed relative to that threshold, which can be met by catching a subset of (often easier) coals.

Also, as in prior years, in some cases, test cases were assessed by TAs who gave feedback manually, rather than through an automated gold/coal process.

6.3 Effectiveness of Sweeps in Peer Review

The research questions in section 6.1 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 3.4 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.

6.3.1 Study Logistics

Courses and Population In this study, we use a corpus of data collected from multiple assignments within 3 courses across 2 institutions (both in the United States). Table 6.1 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 in each course that used in-flow peer review of sweeps. CSPL and CS1.5 were taught at the same institution.

Assignment and Reviewing Workflow Assignment length varied from 4-5 days to two weeks, but most assignments were a week long. For week-long assignments, students submitted sweeps by midnight on the second day after the assignment was released (so, for example, an assignment released Friday and due the

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

Table 6.1: Courses with assignments involved in the study

following Thursday midnight would have sweeps due by Sunday midnight). When assignment duration varied, the sweep submission was roughly 1/3 of the total assignment length.

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 any were 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 (if the number of submissions wasn't a multiple of 4, TAs could fill in the missing reviews).

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 reviews they received. First, they could provide feedback on the helpfulness of individual reviews, including flagging them for inappropriate or offensive content (students didn't flag any reviews for being hostile, though they did flag some for having negligible content). Then, at the end of each assignment, students could also fill out a survey that asked how they changed their assignment based on reviews they received, and based on other student solutions 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.

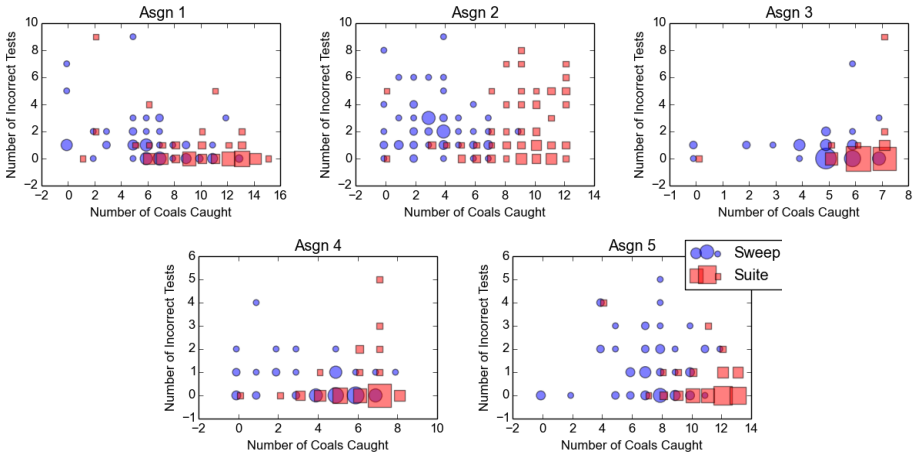


Figure 6.1: Sweep and suite test assessment

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.

6.3.2 Sweep Characteristics

We begin by analyzing the quality of sweeps in isolation of peer review. For peer review to be valuable pedagogically, the artifacts to review need to contain issues for students to comment on, be deep enough for students to find value in reading them, and be diverse enough that students can offer actionable comments to one another. We assess these criteria quantitatively first, then put them in the context of peer review in more qualitative analysis.

We limit our quantitative analysis to assignments from CSPL, which had the most readily automatable gold/coal infrastructure. Our qualitative analyses (section 6.3.3 and section 6.3.7) consider data from all courses.

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 6.1 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 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.

Having established that sweeps are worthy of review, we explore whether they are diverse enough across students to enable valuable peer review. We measure diversity by the distribution of coals detected by sweeps for the same assignment. The sweeps that caught each coal are shown for the same five assignments from CSPL in Figure 6.2. In each plot, each row represents a submission, and each column is a coal. The square is filled if the sweep caught the coal, and empty if not. The student solutions 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 often caught in sweeps. For example, the last "hard" coal on assignment 4 is caught by very few sweeps, while the first "easy" coal is caught by most.
- 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, the figure may be saying more about the lack of diversity in coals 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.

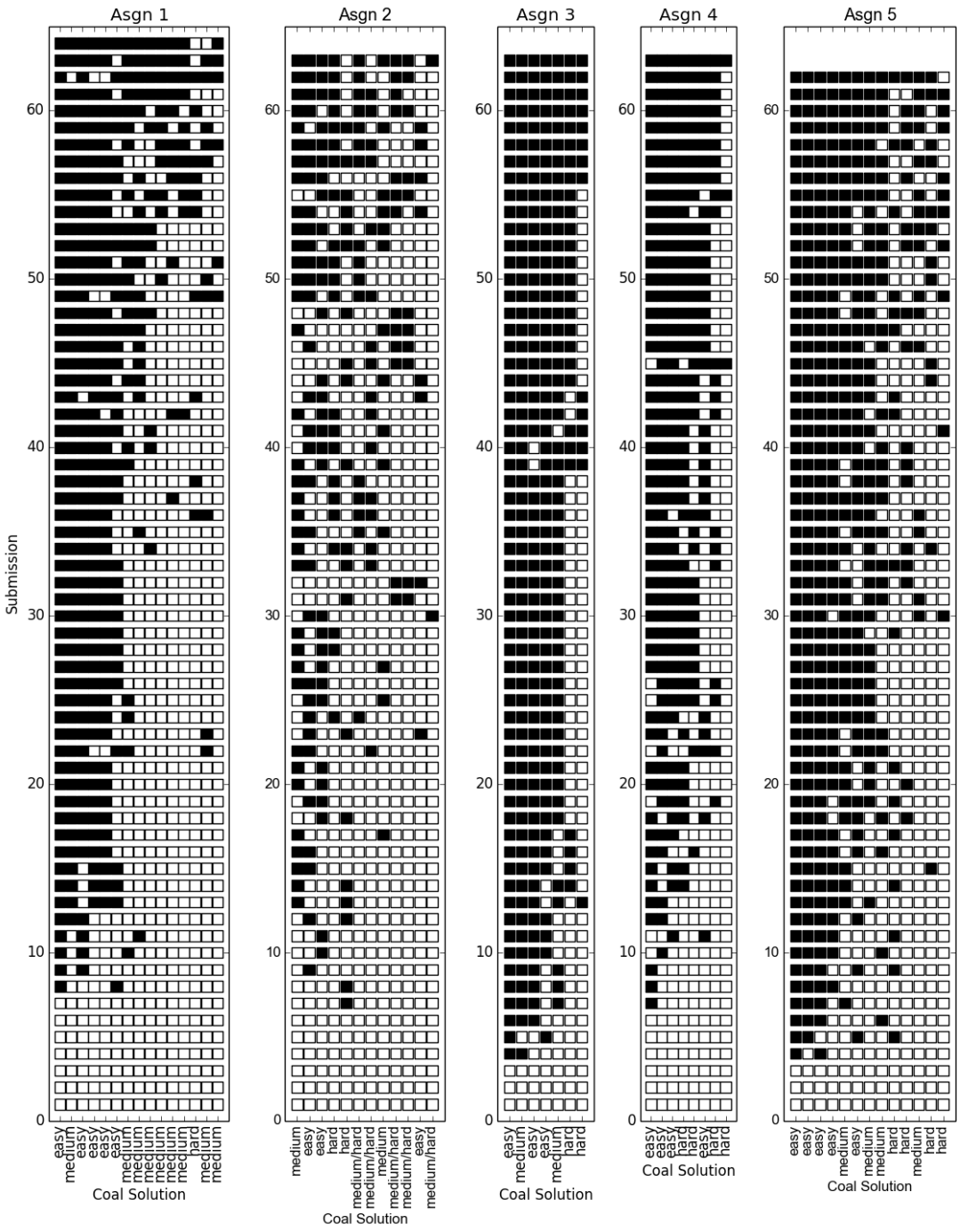


Figure 6.2: Sweep variety measured by coal detection

	both	gained	lost	missed	total
# instances	1633	1011	32	791	3467
seen as both	709	250	2	39	1000
seen as reviewer	182	111	2	71	366
reviewer caught it	160	149	9	76	394
not seen	582	501	19	605	1707

Table 6.2: Coals caught before and after review, in the sweep

6.3.3 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.

6.3.4 Revisiting Test Suite Improvement

Here, we revisit the measure of test suite improvement (section 4.5.4) for a similar selection of five assignments from CSPL in fall 2014, which were the five that used the sweep style of review. The results for the same four categories are shown in Table 6.2. There are several things worth noting about this figure. First, due to the nature of sweeps, there is a much smaller proportion of tests caught in the initial sweep submission, so **both** instances happen with less relative frequency – about half the time, as opposed to the majority. As in the previous data, we see a significant (χ^2=172, p<0.001) difference between the **gained** and **missed** columns relative to review. In particular, over half the cases of **gained** coincide with encountering the solution during review, while less than a quarter do when a coal is **missed** in the sweep and final submission.

There is an interesting contrast between these instances and those from 2013 due to the sweep presenting fewer examples between students, and due to review groups on two of the five assignments. Here, there are many more cases where a student was only exposed to a coal-catching example in the role of reviewer *or* reviewee, rather than as both. In the 2013 data set, students ended up in the **seen as both** situation the vast majority of the time. Here, we see that both cases **seen as reviewer** and **reviewer caught it** appear on their own much more often. In addition, there are proportionally around the same number of **gain**s corresponding to each, so this correlation is explained simply by seeing a test that catches the coal and copying it.

This shows that there is again a correlation between encountering an example during review (directly through seeing it, or indirectly through a reviewer), and the coal being **gained** by the time of final submission. Since this is merely a correlation, we must consider explanations other than the effect of review: perhaps all the students who do well submit at similar times, so they tend to review one another and also tend to catch more coals in their final submissions, for example.

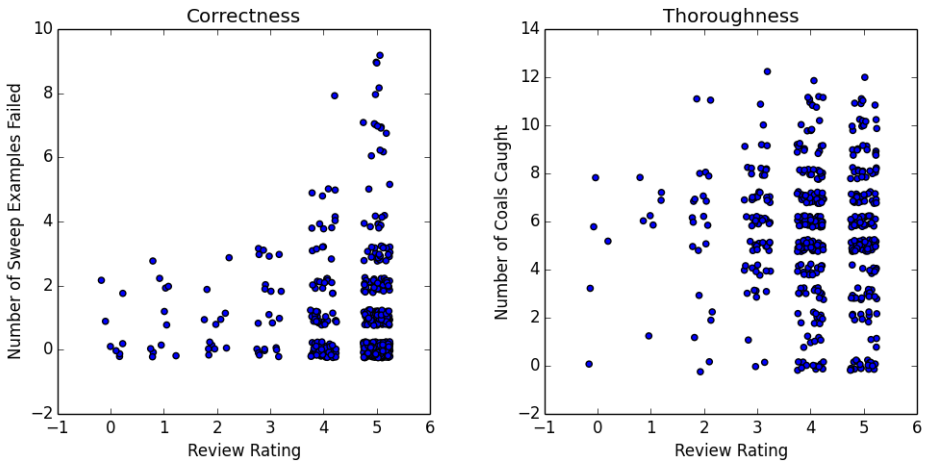


Figure 6.3: Review accuracy revisited

6.3.5 Revisiting Review Accuracy

Here, we revisit the notion of *review accuracy* from section 4.5.2. Since students were evaluated on the same metrics (correctness and thoroughness), and review rubrics had similar questions, it's interesting to see if there are similar trends. Figure 6.3 shows this data for the 5 assignments that used the sweep in CSPL in 2014.

The interpretation of the thoroughness plot is the same as in section 4.5.2: the x-axis indicates increasing agreement from left to right, and the y-axis indicates the number of coals caught by the sweep. Each dot corresponds to a review of a sweep. The correctness plot is slightly different. Since sweeps have relatively few tests in them, it's easy to ask a more exact metric than the *percentage* of correct tests. So in the correctness plot, the y-axis shows the *number* of failed tests. That means that dots higher up indicate a sweep that performed worse, because more of its examples were incorrect.

Thoroughness is, similar to the prior year, fairly diverse across reviews and not strongly correlated with the number of coals caught. We suspect that the reasons are the same, and the correlation is made even less likely by the sweep. The sweep prompt asks if the examples are "important and interesting," which is not only subjective, but not necessarily related to the coals we chose.

Correntness shows more of a contrast with the prior year. In particular, and somewhat troublingly, there are many reviews that indicate sweeps were *correct* when there were several incorrect tests (top right of the figure). There are a few possible explanations for this beyond students simply clicking through reviews. First, the corresponding Likert question changed across assignments. Originally, it asked students to judge if the sweep's examples "correctly reflected the desired behavior." Later, it asked them to count the number of incorrect tests explicitly. However, when asked to count, this changed the end of the scale that indicated a good sweep from the rightmost button to the leftmost.[3] As a result, some of the shown discrepancy may be

[3]We correct for this in the figure by reversing the sense of the scores students gave when given that prompt.

	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)	

Table 6.3: Results of coding reviews of sweeps

simply due to interface confusion. In addition, students had more facilities for giving feedback in the interface in 2014; most importantly, they could give line-by-line comments on tests (this is detailed in chapter 7). We see that students *did* give feedback on incorrect tests in the next section (and often did so through line comments), so the Likert scale may have simply been an afterthought in students' reviews.

From the two years of data we report on here, we think that the prompts are useful for the open-response feedback they provoke (which we discuss in more detail in section 6.3.6), but that the Likert format for responses doesn't add much value.

There are a few ways we envision using quantitative responses like multiple-choice, based on this experience:

- Provide an interface where for *each* example, the student has to indicate if it is correct or not. This would force students to avoid summarizing with a single answer, provide a place where they could describe what they think is wrong, and also enable automated feedback on if the *review* is correct.
- For thoroughness, we could make thoroughness reviewing more concrete by describing in English features we expect (as instructors) should be tested in sweeps, and asking students to identify if they are. For example, we might prompt with "Is there an example that covers identifier shadowing?" in an interpreter assignment that considers variable scoping, and expect the reviewer to be able to identify which test covers that case. If not, they could suggest a test that would cover it. This is related to an assignment style that Jay McCarthy uses in peer grading (personal communication).

As section 6.3.6 shows, the open-ended feedback students give is useful. Making the quantitative feedback more useful likely requires more detailed quantifiable questions in the rubric.

6.3.6 Review Content Analysis

We manually analyzed a random sample of 1088 reviews across the 3 courses to assess whether their contents were constructive or actionable. 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 iteratively developed a category-based rubric for coding reviews, testing for inter-coder reliability on each iteration and refining the categories as necessary. Ultimately, we settled on the following seven categories (the full rubric is in "Review Content, 2014" on page 65):

- **(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 general programming advice about how to use features of the language, improve test structure, or give another tip not directly related to the problem specification or sweep?
- **(REV) Reviewer Gained/Confirmed Understanding** – Did the reviewer indicate that they learned from performing the review, or confirm their own understanding?

We originally distinguished between ERR and SPEC in a different way: ERR was for programming errors that were not problem-specific (like misuse of a library function or typos), and a category like SPEC was specifically for incorrect test cases. Distinguishing the two proved too difficult for the coders, since, for example, a typo in the input to a test is very difficult to tell apart from an actually incorrect test written by a student out of misunderstanding. We chose instead to simply code for instances where the reviewer believes they see a mistake, which is much easier to identify, and separately code for if the reviewer referenced the desired behavior or assignment specification.

We initially had a distinct coding category called LGTM[4] that indicated a review with no content other than a mild compliment like "Nice job" or "Good coverage." The LGTM option was defined to be exclusive to the others, which was awkward when also coding for POS, which seemed like it should be selected for

[4] An abbreviation for "Looks good to me". This is often used in code reviews as a placeholder to get around restrictions on empty reviews. In some companies, it is viewed as a lazy review that does not indicate whether or not the reviewer actually read and analyzed the submission.

some mild compliments. This motivated us to switch to the 0-2 scale for POS, where 0 is for reviews with a completely neutral tone, 1 is for mildly positive comments like in LGTM reviews, and 2 is for specific or especially effusive compliments. We discuss LGTM reviews more below.

At one point, we tried to differentiation between high-level strategy and low-level coding tips, and had a category in addition to TIP. It was difficult for our coders to agree on where the line was between nonfunctional structure changes and big strategic ideas. As a result, TIP simply codes for all programming and code style advice that isn't explicitly pointing out a mistake in test cases.

The remaining categories, POS, NEG, and TEST, required little modification over the different rubric iterations. However, we note that we had difficulty agreeing on POS – the α score was quite low. The line between a review like "These tests are so amazing! Love the way you wrote the helper to generate lists." and "Nice list tests!" is quite blurry for coding. The presence or absence of an exclamation mark can change the score in some cases for some coders. Ignoring tone, which is subjective to code for, and instead having a well-specified category like Specific Compliment may be the right choice in future coding efforts.

Data Assessment and Rubric Agreement Table 6.3 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 as calculated per category across 3 coders (each involved in at least one of the study courses) on a subset of 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 in reviewing. This figure shows that students actually *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 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, so this is an upper bound on the number of LGTM reviews). Though some reviews are LGTM by our measure, these are still fewer than half. 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.

6.3.7 Review Reflection Analysis

In addition to participating in review, students were asked to report on how review impacted their assignments. The prompts were selected depending on how they interacted with review. The first was one of:

- What changes, if any, did you make as a result of receiving reviews on your test cases?
- On what aspects of the assignment, if any, could you have benefited from reading others' tests and giving feedback?

The choice of prompt was made based on if they received reviews or not.
The second question was one of:

- What changes, if any, did you make as a result of writing reviews and seeing others' test cases?
- On what aspects of the assignment, if any, could you have benefited from receiving reviews?

with the choice made based on if the student wrote reviews or not.

Three coders—the author of this document, and two of the course staff for CSPL in 2014—manually coded all of the responses for the prompts where students participated (the "what changed" prompts). The

	CS1.5				CSPL				CS2				α	
	Rev		Rec		Rev		Rec		Rev		Rec		Rev	Rec
EMPTY	6%	11	2%	4	10%	25	9%	23	5%	3	8%	4	**1.00**	**1.00**
NOCHANGE	34%	66	30%	58	25%	65	26%	64	31%	18	38%	20	**0.71**	**0.90**
PROC	3%	6	6%	11	3%	7	9%	22	14%	8	13%	7	0.53	0.62
BUGFIX	5%	9	18%	35	4%	10	12%	31	7%	4	17%	9	**0.71**	0.62
TEST	39%	75	36%	69	52%	132	39%	96	36%	21	25%	13	**0.86**	**0.86**
CODE	8%	16	3%	5	3%	7	2%	4	10%	6	0%	0	-0.00	0.47
LEARN	18%	35	16%	31	11%	28	4%	11	22%	13	4%	2	0.62	**0.68**
DISAGREE	2%	4	5%	10	2%	4	8%	19	5%	3	8%	4	0.47	0.50
NONFUNC	5%	10	8%	15	7%	19	11%	27	7%	4	8%	4	**0.94**	**1.00**
FEATURE	22%	42	28%	53	16%	40	19%	48	21%	12	4%	2	0.65	0.58
Total		193		191		256		248		58		52		
Only		53		51		62		54		53		47		

Table 6.4: Review reflection coding results

two coders other than the author were different than the coders for the review analysis. The goal was to identify what kinds of impact the review process had; or at least that students *believed* it had.

A Rubric for Evaluating Self-Evaluations As with the review coding, the coders iterated on a rubric until reasonable agreement was reached, and then coded the set with an overlapping sample to calculate agreement.

The categories were:

- **EMPTY** – The feedback form was submitted with no text
- **NOCHANGE** – The student explicitly indicated that no changes were made. The coding rubric took "not much" literally, and a self-evaluation indicating that "not much" was changed was *not* labelled as NOCHANGE. The lack of strong agreement here is due to borderling cases like "No changes except for I thought about some new cases." Here, it's not clear if the student actually added the new cases or not, the coders sometimes disagreed.
- **(PROC) Procedural Remark** – The student made a procedural comment about the review workflow, like "I didn't get my reviews until after I submitted."
- **(BUGFIX) Fixed a Problem** – The student indicated that they fixed a problem with one of their tests.
- **(TEST) Added a Test** – The student *added* a test.
- **(CODE) Changed Code** – The student changed their code (not their tests) in some way.
- **(LEARN) Enhanced/Gained Understanding** – The student enhanced or gained understanding or confidence through the review process. This includes learning about language features, learning about the assignment, or being reassured that a test was correct.

- **(DISAGREE) Found Process Unhelpful/Incorrect** – The student thought the review they received, or the test code they saw, was wrong, poor, or unhelpful in some way.
- **(NONFUNC) Non-functional Change** – The student made some nonfunctional change to their tests, like refactoring common expressions into shared variables, adding better comments, or changing variable names.
- **(FEATURE) Discussed Assignment Features** – The self evaluation mentions a specific feature of the assignment.

The full rubric, with descriptions and examples, is in "Review Reflection, 2014" on page 68, along with Figure 9.1, showing the custom coding interface that was built to assist the coders.

Data Assessment and Rubric Agreement Table 6.4 shows the results of coding the self-evaluations according to the rubric. The total number of samples was 659, with an overlap of 100 samples for computing α scores. Counts are broken up by course and by whether they were for performing reviews (Rev) or receiving reviews (Rec). Percentages are given relative to the total number in each category given at the bottom, and Krippendorff's α scores are given as counted across all courses. Bold α values indicate at least tentative agreement (scores above 0.8 are considered strong agreement). As with review coding, when a disagreement occurred, the majority opinion of the three coders is reflected in the count.

Since on different assignments, students may have participated in either or both sides of review, and later assignments had significant overlap, the Only row reports how many of the counts from the column occurred without the other prompt. So, for example, there were 53 instances of a CS1.5 student submitting a prompt for doing review when not submitting a prompt about receiving review. There were 140 instances (193 - 53) in CS1.5 of students being presented with, and responding to, *both* prompts after participating in both sides of review.

Only two assignments in CS2 provided reflection prompts, and both used review groups. Due to a technical error (fixed early on), there were some students shown the wrong prompt, accounting for the 5 overlapping cases.

More iteration on the rubric would have likely yielded better agreement. The CODE category was uncommon enough that it was difficult to iterate on examples for, though it's interesting to consider cases where reviewing *tests* had an effect on *code*. The PROC and DISAGREE categories were subjective and, in retrospect, overly broad. They combined several previous categories, and would have benefited from more iteration before the final coding. The FEATURE category was added before the final coding without much testing, with the coders observing a type of comment that was yet uncoded.

Observations on Review Reflection Mirroring the review analysis, the most common category in review reflection was adding a new test (category TEST), across all three courses. In all three courses, students were more likely to report adding a test from sweeps they saw than from reviewing. Also in all three courses, a significant number of students did report adding a test due to *receiving* reviews, with a smaller proportion in

CS2, which is consistent with the coding for reviews, which reported new tests being suggested about half the time.

The NOCHANGE category is the second most frequent. Between a quarter and a third of the time, students explicitly indicated making no changes during review. This was for a number of reasons, including unhelpful or content-less (perhaps LGTM) reviews, but also students who simply saw little value in the suggestions that were made and decided not to implement them. We also coded as NOCHANGE responses that said things like "my review suggested tests that I already had written but didn't include in my sweep." These percentages (and the relatively low percentages for DISAGREE, PROC, and EMPTY) suggest that students in these courses indicated something clearly positive out of review over half the time, as a rough approximation.

6.4 Copying of Examples

We outlined a specific stance on copying examples in section 3.2.1, where we argue that it's useful for students to learn to make the judgment call of which examples are worth it to copy. One thing we learned through conversation with students was that they often felt uncomfortable using examples they saw in review in their own final test suites. They felt this way despite being explicitly instructed that it was encouraged. This showed up in review reflections, as well, where students would indicate that they "stole" or "took" examples that they saw during review.

Given this, it's useful to ask just how often examples were copied during review. This is difficult to answer in an automated system with certainty, since there can be both false positives—students ending up with similar final tests without the change coming from review—and false negatives—students who copy examples and change their formatting or tweak input data so the resulting test is no longer identical. This is a slightly different scenario than detecting outright plagiarism in the style of industry standard tools like MOSS,[5] but the insight from those tools makes sense in our setting. We note that since CSPL and CS1.5 were taught in Pyret and CS2 in Scala, MOSS does not come with an off-the-shelf tool that applies.

After experimenting with an automated tool of our own design, we reach a few tentative conclusions. First, copying is certainly happening in some cases, and sometimes fairly sizable chunks of sweeps are copied. Second, any automated solution is likely to only be a starting point for an accurate count of copying, and human judgment will be required to identify false positives.

One common strategy for detecting copying is to split the program into tokens (common punctuation, words, symbols, and so on), strip extraneous whitespace, and look for runs of tokens that are the same between files. We can apply this to compare initial sweep files against final test submissions across all students. Table 6.5 shows the results for an assignment from CS1.5 and one from CSPL, looking for runs of 50 identical tokens.

That is, there are 22 and 7, respectively for the two assignments, sweep/final submission pairs that have more than 30 matches by this metric – large chunks of the sweep were copied. Cases with this many matches are pretty clearly cases of copying, but the shorter matches require human judgment. For example, some of

[5] `https://theory.stanford.edu/~aiken/moss/`

# of matches	CSPL Asgn	CS1.5 Asgn
1	4	6
2	0	4
3-10	6	5
10-30	10	0
>30	22	7

Table 6.5: Example of code similarity metrics in two sample assignments

the shorter matches appear to be reasonable minor tweaks to helpful starter code given out with the sweep file template. It's not immediately clear if that's a case of copying, or if two students simply made the same tweak to a number literal. Upon visual inspection, in the CSPL assignment, some of the cases up to around 10 matches are clearly boilerplate with tweaks, but also contain some instances of copying.

In addition, this is just one configuration of parameters. Doing a different analysis that searches for entire duplicated lines, and with a less stringent limit on length, finds other similarities. These are too short to match 50 tokens in a row, and also require human judgment. For example, using a line-based comparison, this line is similar between two files, with no other similarities detected:

```
deleted = update(c, lam(t): mt end)
```

Without getting deep into the context this line appears in, and with some understanding of the assignment, this is *probably* not a case of copying; it's a reasonable variable name and function call to have as an intermediate step in many tests. However, similar short matches could easily indicate copying if students got an idea for a short base-case test from another student and moved it over. This suggests that the 50 token approach may have false negatives as well as false positives, since it didn't flag this line. Lowering the required token length detects this case, but increases the long tail of potential matches. As a result, it's potentially misleading to simply pick a setting and present the raw data.

Instead, we suggest a way forward. Trying several of these strategies shows that there are a handful of pairs on each assignment with a lot of clearly copied code, and a long tail that requires human judgment. The right way to tackle this analysis is to use the automated system to get a starting point for human coders. From the automated analysis, we can generate an interface similar to a difftool, showing the potential copying cases side-by-side. A human coder can use that interface to make an assessment about copying, and multiple coders can be used to gain confidence that a judgment is correct. This is in line with the general workflow of tools like MOSS, which are to be used to prioritize and flag *potential* cases of plagiarism,[6] which require human investigation This manual effort is beyond the scope of this study, and is left as a useful future exploration.

[6]It's worth reiterating that in our case, we simply are looking for potential cases of *copying* tests, which is a positive action to be encouraged, not penalized.

Chapter 7

Tooling for In-flow Peer Review

All of the studies mentioned were supported by Web-based software that enabled the review workflow. There were two major versions of the software used. One was used in the initial study in 2013, which was Pyret-specific and had feature support beyond simply peer review. The other was language-agnostic, supplanted the original system for the 2014 courses, and is in use at other institutions. This chapter discusses the two versions of the tool with examples of their configuration and use.

7.1 Captain Teach 1.0

This section reports on work done in collaboration with Daniel Patterson, Shriram Krishnamurthi, and Kathi Fisler [19].

The first version of Captain Teach combined several different learning environment features from a student's point of view:

- A rich editor code window for writing Pyret code, on the left-hand side.
- An interactive REPL for Pyret code, on the right-hand side.
- A review mode for the code window that students could enter into between submission stages.

Assignments were built by writing a combination of prose and configuration text in a custom markup language built on top of Scribble.[1] The configuration language could specify that one or more reviewable code areas should be rendered on the page and how many review should be done. In addition, the markup could include prose and images describing the assignment. The interactions area was always present, and would act with the definitions of the most-recently-run code area.

The most relevant feature relative to this document is the combined coding and review interface students were shown. In order to enforce that students work on examples first, they were given an editor where all

[1] `https://docs.racket-lang.org/scribble`

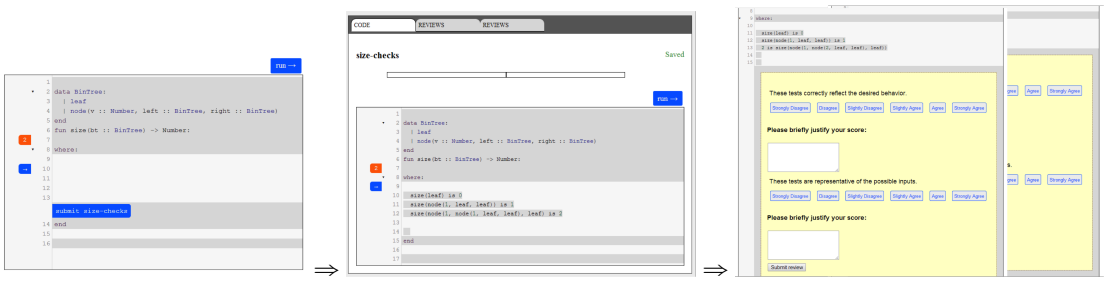


Figure 7.1: Captain Teach 1.0, student workflow

code except for the examples block was grayed out and uneditable. They could edit within the test block (denoted by `where: ... end` in Pyret), and submit when they were ready. Immediately upon submission, the whole editor would become uneditable, and they would be presented with several tabs containing code to review. They could click on these tabs to see the review interface, and complete the reviews in any order. Once the review was completed, they could move on to complete the next step of the assignment, usually the body of the function that was just tested. A screenshot of this workflow is shown in Figure 7.1.

There is a similar workflow for submitting code and data definitions, with corresponding review steps. This system was built specifically to support review of the Design Recipe [12], which includes each of these activities as intermediate stages. In addition, the editable and grayed-out areas were involved Pyret-specific setup in the software, due to Pyret's specific syntactic support for things like test cases attached to function definitions with `where:`.

7.1.1 Seeded Reviews

We used staff-written solutions in two ways in this system. First, in order to provide the first students to submit with something for review, there needed to be some work in the system to review up front. So, there were a small number (usually just equal to the number of reviews) of solutions that would be uploaded ahead of time. Second, in order to discourage students from writing cursory reviews, the course staff also wrote one known-good and one known-bad implementation. Students would randomly (with 50% probability) be assigned a known review when choosing their review assignments. Again with 50% probability, they would be shown either the known-good or known-bad implementation. If they gave a Likert rating that was negative on the good solutions, or positive on the bad solutions, the system showed them a pop-up message warning them of their inaccuracy. There was no grade value assigned to this, it was simply a nudge to the students that they may have missed something.

7.1.2 Review Prompts

Since the system was set up primarily for the review of Pyret code, review prompts were hard-coded to questions about data structures, tests, and code. Each prompt had two questions, each with a Likert component

and a free-response component. In all cases, the first question was targeted at objective criteria, and the second a subjective one. The prompts were:

- *Tests* - "These tests are representative of the possible inputs."
- *Tests* - "These tests correctly reflect the desired behavior."
- *Code* - "This code correctly implements the desired behavior."
- *Code* - "This code is structured well."
- *Data* - "This data definition covers all the cases of data that it needs to represent."
- *Data* - "This data definition is structured well and all the names are appropriate."

This system is no longer actively deployed and maintained, but the code is open-source and available at `https://github.com/brownplt/captain-teach`.

7.2 Captain Teach 2.0

This section reports on (unpublished) work done in collaboration with Arjun Guha, Joseph M. Collard, Shriram Krishnamurthi, and Kathi Fisler.

The design of the first Captain Teach system, and our initial studies, suggests several ideas for an improved system. First, the system simply cannot support instructors teaching non-Pyret courses. Second, the layout of the first version of Captain Teach is limited to single-file programs, which is an unfortunate artificial restriction on assignments. Third, the workflows permitted by the first version didn't admit much experimentation with workflow through the assignment or with trying out different rubric questions. For these reasons, we built a new system to handle in-flow peer review that could tackle some of these more general challenges.

7.2.1 Workflows

There are many possible use cases for in-flow peer review (a number are explored in [7]). Some simple examples are:

- Students submit tests, then are assigned several reviews, then submit code and updated tests for a final submission, and finally submit a reflection on how review was helpful. The review rubric is customized for each assignment to instruct students to look for particular features in the test suite.
- Students are working on a multi-week software engineering project. Reviewing the whole thing would be too large a task. So the process is broken up into milestones: design documents, test strategy, user interface wireframes, and finally pieces of the implementation. Each of these is reviewed by several other groups.

- During lab time, students are asked to switch between reading and writing code 3-4 times per session. This can quickly disseminate ideas around the class if students are given open-ended tasks, and aren't all expected to have the same solution. To support this, the instructor sets up several stages of submission and review ahead of time.

It is easy to imagine other kinds of assignments that require very different workflows. A new workflow may have any number of milestones, it may ask students to use a new rubric for reviews, it may group students in different ways, and so on. A useful in-flow peer review tool should support these workflows and more.

7.2.2 Creating and Managing Workflows

The wide variety of workflows possible demands a usable interface for instructors to configure different assignments with ease. Figure 7.2 shows the interface instructors can use to create assignment workflows.

The primary components of the assignment creation workflow are:

- Choosing assignment milestones.
- Configuring response tasks—reviews and surveys—for milestones.

All of these are configurable options in CT2 through both a visual interface and through a textual markup language.

Assignment Milestones Assignment stages are explicit steps where students submit artifacts like code or tests. An assignment author can choose any number of milestones, which have simple configurations like submission instructions and a name. For student-submitted artifacts, the current interface simply expects students to upload a zip file.[2]

Response Tasks There are three kinds of responses that an instructor can configure for a milestone: **peer reviews**, **controlled reviews**, and **surveys**. For peer reviews, the instructor chooses both how many reviews should happen, and what rubric should be presented (which we show in section 7.2.3). Rubrics can include typical form components, like text responses and checkboxes. Controlled reviews are like peer reviews, but the instructor provides the solution rather than the student. This can be useful for providing a calibration round of reviews to train students on submissions of known quality (as in [5]). Surveys are configured similarly, but instead of causing reviews to be assigned between students at the given milestone, they prompt the *submitting student* to give feedback that is visible to the instructor.

These two kinds of tasks allow for quick configuration of linear assignment workflows that intersperse submission, review, and self reflection.

7.2.3 Student Experience

There are a few elements of the student experience in CT2 that were honed over time in response to classroom use and student suggestions.

[2] A prototype of including a collection of files from Google Drive is built but not deployed

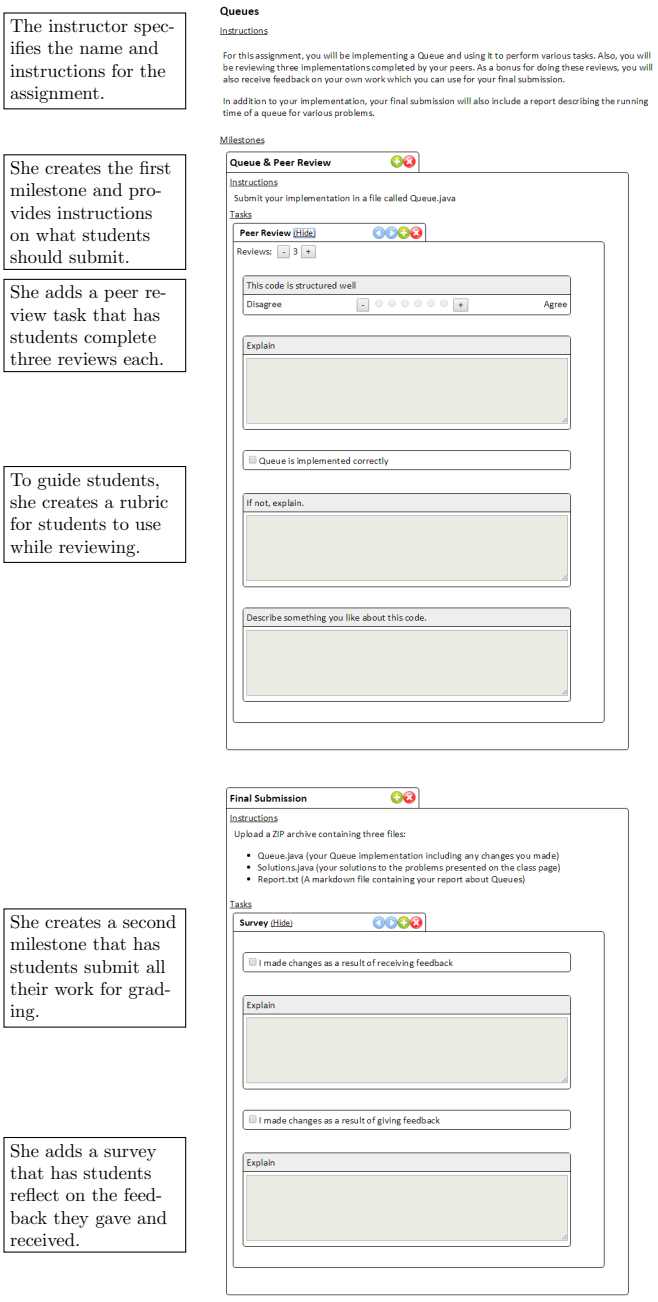


Figure 7.2: Point and click assignment building

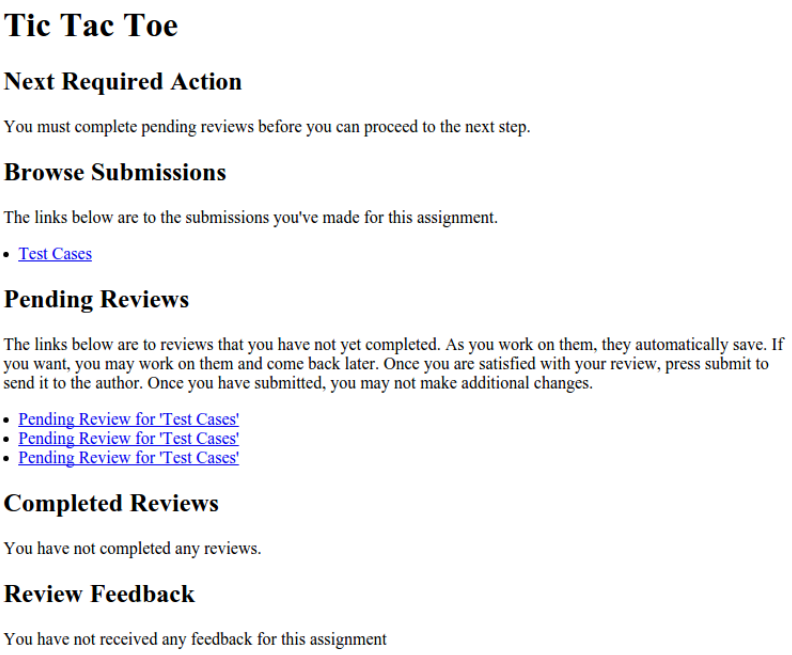


Figure 7.3: The student task list interface.

Explicit Task Tracking In an in-flow assignment, students need to know what to do next. Especially on the first few assignments, it's not always clear whether reviewing or the next submission needs to happen next, or when a student has completely finished the assignment. In addition, students need a way to get back to feedback they've received and access reviews they've written. The student assignment summary in Figure 7.3 details this information for students, so they can access the milestones they've completed, and know what to work on next.

Soft Submissions Often, students want to submit an initial version of their work, but not have it published for review, so they can refine it before the deadline while having the assurance that some stable version has been uploaded. CT2 makes a distinction between the *upload* and *publish* steps for a milestone. This lets students preview their work that will be reviewed, and update it if necessary. At the milestone deadline, their most recently uploaded submission is published if they have not explicitly published one, so that they will not be late. The upload/publish distinction is important in particular for assignments with review, because once a mileston has been published, the published version cannot change, since it will be assigned for review.

Review Interface The student code review interface is shown in Figure 7.4. The left side shows a current file, and an entire directory can be navigated by using the directory links at the top, as CT2 supports submission of entire directories as well as single files. The instructor-configured rubric appears on the right-hand side. On the left, any code display window has clickable numbers in the left column for inserting an inline

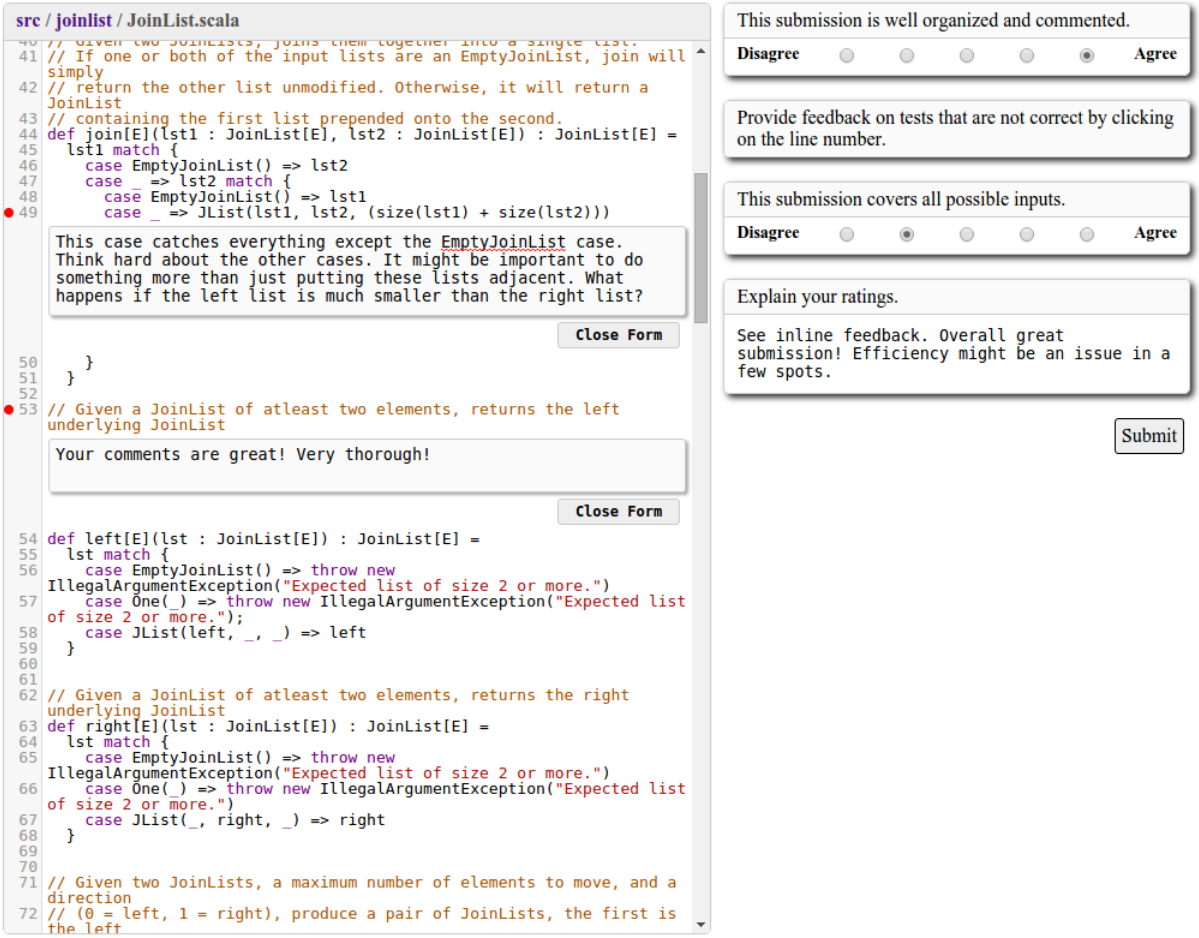


Figure 7.4: The student review interface.

comment, similar to the code commenting interface in tools like Github. This allows for a much more natural feedback style than referring to line numbers in prose, and introduces a common professional interface for code review. The entire review is frequently auto-saved so a student can draft and refine their review over time. Feedback is shared with the reviewee and made immutable when "Submit" is clicked.

Reviews-of-reviews and Flagging Reviews can be useless or problematic for several reasons. A student may have submitted something incomplete or nearly empty for their review, whether accidentally or through lack of effort. A student may write hostile or inappropriate comments that are pejorative to the reviewee. A review may be confusing or not describe problems well.

To help address these cases, CT2 provides a space for reviewees to give review feedback that is visible only to instructors, and to *flag* reviews that are inappropriate. Flagged reviews show up with an explicit annotation in the instructor dashboard (Figure 7.5) so instructors can quickly tell if something is amiss. In our experience, students have only flagged reviews for having little or no content, and haven't flagged them for inappropriate content. Our coding of a significant sample of reviews revealed a single overly hostile review, and the readers of the review disagreed (section 6.3.6). So while we haven't seen especially problematic cases happen yet, CT2 provides a quick way for instructors to be notified if it does.

7.2.4 Instructor Dashboard

A happy consequence of in-flow peer review is that it forces students to work on intermediate milestones for the duration of an assignment; they cannot scramble to finish everything the night before it is due. CT2 provides a dashboard for instructors to monitor the progress of the whole class as well as individual students. The instructor can quickly check that students are engaged and completing milestones on time, and debug any issues that may arise.

The assignment overview displays the number of students who need to submit to each milestone, the number of completed reviews, and the number of students who have actually viewed their feedback. If these numbers are low, it may indicate that students are not engaged. It could be that they've received several poor-quality reviews or that they received reviews that were too late to be useful and are now disinterested in the peer review process. By discovering these issues early an instructor is able to respond to the needs of their students.

In addition to a quick overview, the dashboard provides a detailed view of each milestone (Figure 7.5). This view provides staff with quick access to individual submissions and reviews. By spending a few minutes browsing through student submissions on an early milestone, staff may notice a common error among the submissions that should be addressed. For example, an assignment write-up may have been unintentionally misleading or simply incorrect. Similarly, a quick glance at several student reviews might reveal a rubric question is not having the desired effect. These kinds of errors can be swiftly corrected due to the visibility that CT2 provides. We have found this early feedback very useful in our courses.

Finally, despite our best efforts to make CT2 easy to use, students may still have trouble. To help instructors solve problems, CT2 makes it easy for an instructor to assume the role of a student (i.e., like the Unix `sudo` tool). This allows the instructor to immediately see the issue the student is having and reply with

Tic Tac Toe - Test Cases Milestone

Peer Review

Assigned : 90
Completed : 46
Feedback Seen : 32
Clicking a student ID will bring you to the student view for the selected review.

Reviewer ID	Reviewee ID	Completed	Flagged	Feedback Seen	Actions
anonymous@student.edu	anonymous@student.edu	Yes	Yes	Yes	Mark Incomplete
anonymous@student.edu	anonymous@student.edu	Yes		Yes	Mark Incomplete
anonymous@student.edu	anonymous@student.edu	Yes		Yes	Mark Incomplete
anonymous@student.edu	anonymous@student.edu	Yes		Yes	Mark Incomplete
anonymous@student.edu	anonymous@student.edu				Mark Complete
anonymous@student.edu	anonymous@student.edu	Yes			Mark Complete
anonymous@student.edu	anonymous@student.edu	Yes		Yes	Mark Incomplete
anonymous@student.edu	anonymous@student.edu				Mark Complete
anonymous@student.edu	anonymous@student.edu				Mark Complete
anonymous@student.edu	anonymous@student.edu	Yes			Mark Incomplete
anonymous@student.edu	anonymous@student.edu				Mark Complete
anonymous@student.edu	anonymous@student.edu	Yes			Mark Incomplete

Figure 7.5: A live view of submissions-in-progress, including reviews submitted and feedback viewed

stepwise instructions to resolve their problem.

7.2.5 Example Workflows

Here we provide some examples of using CT2 to configure workflows.

Tests (with Control) and Implementation In the CS2 course mentioned in section 6.1, students were asked to implement a Tic Tac Toe-playing agent using the minimax algorithm. The assignment was broken into two milestones. The first mile- stone instructed students to submit a test suite for their agent. After submitting their test suite, a student was given three tasks: two anonymous peer reviews and one controlled review, configured to resemble the other anonymous reviews. All reviews used the same rubric containing guiding ques- tions asking the students to evaluate the exhaustiveness of the test-suite, report any errors found, and to provide feed- back on the programming style and structure of the code. A test-suite with known errors and poor coding style was provided for the controlled to evaluate students' reviewing ability.

The second milestone directed students to submit the whole assignment including any changes they made to their test suite. Once their final submission was received, they were given a survey asking them to reflect on the changes they made as a result of giving and receiving feedback.

Reviewing Groups In order to support the review group structure from chapter 5, CT2 was extended with an additional task type. This allowed the instructor to specify, along with the rubric for a review activity, a review assignment configuration file. This file specified which students were in the no review, gets review, and does review groups.

In addition, this task type allowed surveys to be conditioned on which review group students were in. So the final surveys for this task type changed depending on if a student participated in review. Students who received reviews got a prompt asking how the feedback changed their final solution, and students who didn't receive review got a prompt asking them to speculate on how they could have benefited from review. There was an analogous pair of questions for students who wrote, or did not write, reviews.

Opt-out In order to support the opt-out structure from section 5.4, we (ab)used the review groups infrastructure. For each opt out assignment, we created two assignments, one that skipped the review step entirely, and another that assigned all students to *both* the does review and gets review group. Students chose at submission time which assignment's flow to use.

This feature would be nice to make a first-class member of the system. That said it was gratifying that the existing tools in the system could provide this unexpected reuse in a short time. We expect that adding a few more key configuration options will support even more useful combinations for unexpected situations in the future.

7.2.6 Implementation

There are several decisions to make with a Web-based service like CT2. There are two main relevant choices in the design of CT2. First, it uses standard cloud-based tools as much as possible, to enable easy deployment

on services like Google Cloud Hosting and Amazon Web Services. This includes deploying from a standard Docker container, and using the de facto standard of Amazon S3's API for storing and accessing files. Second, CT2 avoids storing user credentials as much as possible by using third party identity providers via OpenID. More and more institutions have access to Google Apps for Education, allowing CT2 to link to existing student accounts and avoid the security concerns of managing student passwords.

Chapter 8

Looking Back and Forward

The studies reported in this document explore peer review of tests early on in programming assignments. They demonstrate the promise of the workflow for improving students test suites and sweeps, and serving as a mechanism for communication between students during assignments. It leaves a number of questions for future work, and questions about the long-term benefits of the workflow.

8.1 Retrospective Survey

To understand how students benefited beyond what we see in course data, in spring 2016 we surveyed students who had used used in-flow peer review of tests over the past three years. In particular, we surveyed students from CSPL and CS1.5 in 2103, 2014, and 2015 (the 2015 classes' data are not included in the studies in this document).

A screenshot of the complete survey is shown in "Appendix C – Retrospective Survey" on page 78. We briefly summarize the data here. Because we believe that code review and testing are both skills that ought to be important in other programming settings, especially industry, we were curious if students would find that having participated in test review would be more helpful as time went on. For each year, we asked a pair of questions aimed at getting a sense of that (with year changed as appropriate):

- The peer review process we used in fall 2013 was useful.
- Looking back, has your feeling of the process's usefulness changed from how you felt when you participated in fall 2013?

For 2014 and 2015, we also asked with what frequency students chose to opt-out of review, since students opting out were likely to have found the process unhelpful. Since some students took both courses in different years, the survey was presented with one section per year, and students could respond to the questions for the years they pariticipated.

We asked three additional questions that weren't year-specific:

- Of the courses you've taken since, in which would test review have been helpful (if any)?

Course/Year	Responses	Useful?					Changed since?				
		RD	SD	NU	SA	RA	ML	SL	NC	SM	MM
CSPL/2013	3	0	0	2	1	0	0	0	0	3	0
CS1.5/2013	14	1	2	2	6	3	1	1	5	4	3
CSPL/2014	11	0	2	0	7	2	0	0	9	1	0
CS1.5/2014	24	0	1	3	16	4	0	2	13	7	2
CSPL/2015	6	0	2	0	7	2	0	0	5	1	0
CS1.5/2015	26	0	2	1	18	5	0	1	19	6	0

(RD=Strongly Disagree, SD=Slightly Disagree, NU=Neutral, SA=Slightly Disagree, RA=Strongly Agree)
(ML=Much less useful, SL=Somewhat less, NC=No change, SM=Somewhat more, MM=Much more)

Table 8.1: Quantitative results from retrospective survey

- Have you used concepts or skills you learned from peer review of tests in other contexts?
  - Yes, in other coursework
  - Yes, in internships
  - Yes, in in research
  - Yes, in personal projects
  - Other
- Use this space to elaborate on any of your responses above, or give other open ended feedback:

The answers for the quantitative questions is summarized in Table 8.1. Most students across both classes and all three years slightly agreed that peer review was useful, and most didn't have their opinion change significantly in the time since they participated. There is a slight suggestion from based on the proportions in the changing opinions question that, as time goes on, more students find increasing utility. Self-selection bias of the students who chose to take the survey could also explain this.

One of the motivations for both peer review and testing is their relevance to industry.[1] 20 of the 78 respondants reported using the skills they learned in internships. In addition, those 20 respondants didn't include the 5 students whose opinion of utility diminished over time. 22 students reported using the skills they learned in other coursework, and across students, 11 other courses were listed as being potential useful settings for test review. Students especially mentioned software engineering and introduction to computer systems, mentioned 7 and 9 times, respectively. The rest of the mentions of courses were spread across mostly upper-level courses, and a few intermediate introductory courses.

This data is unsurprising but reassuring as a defense of the utility of in-flow review of tests. It reflects the general, somewhat mild, approval that students had with the process, with some occasional specific issues.

[1] As just one example, Gavin Andresen, lead maintainer of the Bitcoin codebase, said recently that "Our number one problem in Bitcoin is code review." (Personal communication with Shriram Krishnamurthi)

It also shows that students (at least when prompted) find that the skills they learned from the process have transferred to other contexts.

8.2 Future Directions

There are a number of productive directions to take the ideas presented in this dissertation for future studies.

8.2.1 Opportunities for Test Review

Within the space of test review, there are a few clear next steps.

Application to Software Engineering A large part of the goal of in-flow peer review of examples is to identify misconceptions and holes in problem understanding early on. This can course-correct an implementation effort before costly effort has been spent on incorrect code. This outcome is beneficial in constructing software in general, not just in programming assignments in undergraduate courses. The idea of testing early and discussing tests is fundamental in extreme programming [2]. In that context, the discussion of early tests is more likely to happen in the context of pair programming than a separate code review step.

It's natural to ask if a version of the sweep review process could be useful as part of a general programming workflow. For example, multiple developers could each write an independent set of examples, and then take the union of the tests after discussing and resolving inconsistencies. This would require adapting the workflow for particular projects and teams, and designing ways to evaluate how well it worked. In a larger-scale software setting, it's unlikely that it would be feasible to write multiple coal implementations for the purpose of test suite evaluation, for instance. Other techniques like mutation testing [9] could be valuable here, along with more traditional measures like defect rates and time to project completion.

Alternate Rubric Design The rubrics used in the studies presented here were quite simple, and focused on subjective questions with an agreement score. In the final iteration, we asked the first objective quantitative question – "How many tests are incorrect?"(discussed in section 6.3.5). Since it was introduced later on in the semester it may have simply caused confusion as it changed an existing rubric template.

However, there are many more opportunities for structured rubrics and quantitative questions. For example, providing an interface where students can indicate on each test whether or not they believe it is correct, and provide a fix if incorrect, could allow for much more rigorous checking of review accuracy. Similarly, we could provide rubrics that concretize thoroughness more. For example, we could provide snippets of an implementation and ask students if the sweep they are reviewing would exercise particular portions of the code. We could show other tests and ask "did the student have a test like this one?" All of these have trade-offs between learning, giving away too much of the solution, and benefits for measuring review quality and student engagement.

Alternate Review Assignment We designed Captain Teach to assign reviews as soon as possible, so that students who wanted to work on reviewing immediately after submitting sweeps could do so as often as

possible. This means that the students' schedule and submission habits could impact which sweeps they saw during review. A random assignment could alleviate this, but would require waiting for more submissions to come in to build up a random pool of submissions to assign from.

If we are willing to delay review assignment, there's a potentially better method. Given that we can calculate the data in Figure 6.2 automatically, we could wait until all sweep submissions arrived, and then assign students to review one another based on characteristics of their sweeps. For example, we could maximize the average difference between coals caught in each review pairing, so that students have a chance to see more different ideas. We could also experiment with exposing students with ostensibly weaker sweeps (catching fewer coals) to sweeps that catch many coals. There are several options available, but the key idea is to use an objective metric to come up with a more useful assignment of reviewer and reviewee than random or time-based.

8.2.2 Other Review Targets

The studies reported in this document reflect a focus on review of tests, and even more narrowly, review of *examples* in the sweep. The idea of in-flow peer review applies to much more than just tests [7], and workflows like the design recipe [12] provide more opportunities for review. In 2013, the first pilot of in-flow peer review using Captain Teach, we touched briefly on several other artifacts for review, but did not study them in detail. Each provides interesting future opportunities for study.

Review of Code In the 2013 courses, students did review of tests and code on each assignment, adding another workflow step. Because of the arguments in section 4.5.1, we simplified the workflow to just tests in future years. However, this was an observation made relative to particular courses, while trying to minimize the impact of peer review on the schedule. For longer assignments or different curricular goals, having a code review step in addition to or instead of test review could make a lot of sense. We observed in reviews of tests that students learned, for example, coding organization tips (the TIP category in section 6.3.6 and the NONFUNC category in section 6.3.7). It's quite possible that much more of this type of learning would result from code review.

Pedagogic code review in various forms is more well-studied in the literature (chapter 9), as well. Future directions could leverage that understanding, and explore how existing code review strategies could adapt to the in-flow context, and complement review of other artifacts, like tests.

Review of Data Structures In 2013, one assignment in CS1.5 did code review of a data structure design. The assignment is to implement a tree zipper, and a key component of the assignment is to pick a data structure representing a location in the tree that could efficiently express a number of operations.[2]

We introduced a round of review in which students submitted the data structure with comments, but no associated code. Anecdotally, students reported that it was difficult to write down the data structure without implementing some of the operations to experiment with how it would be used. In addition, students who

[2]The assignment was a version of this one: `https://cs.brown.edu/courses/cs019/2014/Assignments/updater.html`

did a lot of work up front submitted data structures that could "give away" the answer to other students. Depending on the assignment and course context, this may or may not be desired.

The problem of writing a data structure without iteratively refining it along with an implementation is similar to the problem of implementing an exhaustive test suite before having an implementation. Our response to the issue in the case of testing was to design the sweep, which required more conceptual effort and less attention to irrelevant detail. It's interesting to consider what the analog of the sweep is for data structure design. The primary goal would be to allow students to submit reasonable sketches of data layout, without having to design a significant portion of the implementation that uses it first.

One other facet of data structure review that differs from code and test review is that it lacks an obvious objective metric. The studies in this document make use of the specific technique of gold/coal evaluation of tests, and other metrics like code coverage are also applicable to tests. Code can be evaluated with any number of testing strategies. But it is difficult to tell if a data structure will "work" for a particular problem programatically. The lack of such a metric requires much more manual effort in grading and evaluating the quality of these submissions, and in studying the efficacy of reviews.

Review of Written Work CSPL has written assignments, where students argue a point or discuss a programming language design decision in response to a prompt. For example, a question might propose an alternative implementation for a language feature than described in class or the textbook, and ask students to evaluate its merits. In CSPL in 2013, students also performed review on these answers. Their grade was a weighted sum of their initial submission and their submission after incorporating feedback and other students' answers, heavily weighted towards their initial submission.

This review process provides a number of opportunities for students to learn, and indicate their learning. Practice identifying good and bad answers as reviewers is a useful skill in its own right, and a complementary one to authoring answers. Also, reading others' solutions gives students useful perspective on these open-ended questions. If a student is able to effectively revise their solution in response to seeing other answers, then they've demonstrated yet more learning.

Table 4.3 summarized poll results for students where students reported that reviewing written assignments was the most helpful across all the categories. One potential future direction for studying this further would be to use a written design discussion as part of a larger assignment, and give students the opportunity to learn from one another's responses. This may complement other artifacts, like tests, and also provide a way forward for reviewing design decisions on data structures, which may require a more structured plan in prose for how the structure will be used.

Other Artifacts Tests, code, data structures, and written work are the types of artifacts the courses in these studies targeted for review. They all made sense in the context of assignments that were roughly a week long, and could be introduced with reasonable overhead, though combinations of them could become onerous.

There are, of course, many other opportunities for review artifacts beyond these. For example architecture or system diagrams could be useful review artifacts in a software engineering course that has larger-scale projects. A graphics course could use sketches of eventual planned renderings, and students could comment

on the feasibility of accomplishing the rendering; an animation class might similarly make use of storyboards. Storyboarding in general is a useful review artifact for any software application with a user interface.

Broadening the scope to this point simply highlights that many computational problems can make use of many different viewpoints and granularities of intermediate artifacts in a solution. Intermediate stages that are relatively low-effort to produce, and provide strong opportunity for catching mistakes and exploring the problem space early, are great candidates for review.

Chapter 9

Related Work

Peer code review and test-focused programming assignments, on their own, have been studied in detail, but do not focus explicitly on the peer review of tests, as in these proposed studies. The effectiveness of code review has been studied extensively in industry, as well, and can inform the practive of code review we choose in our courses.

9.1 Code Review in Industry

Fagan's seminal work on code inspections in an industrial setting [11] finds that putting inspections at carefully-delineated points throughout a product's life cycle can save time by fixing faults earlier, before other work builds on the buggy code. In Fagan's experiments, there are three inspections: one after an initial design phase, one after initial coding, and one after unit testing and before system-wide testing. Experiments show that maximal productivity is reached by including only the first two inspection steps due to the high cost in developer time relative to the time saved by early detection, but that using the first two steps increases programmer productivity by 23%, according to their metrics.

In Fagan's experiments, the effects of putting the testing step before implementation weren't studied; tests were reviewed only relative to a particular implementation. This was mainly in service of finding defects in an existing implementation before shipping a product, and leaves open the question of what role tests and test review could play in early development, rather than quality assurance.

Fagan's original results are for *formal code inspections* [11], which consist of a meeting of several developers (including the original author), conducted with prior preparation and with a separate *reader*, separate from the author, who presents the work. Defects' cause and detection are documented in detail, which acts as a sort of "rubric" for the code review.

While formal code inspections demonstrably find valuable defects, it is not clear that the organization of a meeting is required in order to have a comparable effect. Votta studied the necessity of meetings for code inspection, and found that the majority of defects—over 90%—were found in the *preparation* for the meeting, rather than in the meeting itself [27]. Votta concludes that much of the benefit of code review can

be had without the overhead of scheduling in-person meetings.

Bacchelli finds that at Microsoft, while defect finding is important, it is only one of several top motivations developers see for review [1]. Also scoring high in developer surveys as motivations for code review are general code improvement, suggesting and finding alternative solutions, knowledge transfer, and team awareness. While performing reviews, developers indicate specific cases where they learned about a new API that they could use in their own code, or provided links to the code author with documentation of other alternatives, suggesting that these activities do indeed occur.

9.2 Peer Code Review in Assignments

Peer review has been used in a number of different contexts in computer science courses. Topping's 1980–96 literature survey on peer review in higher education [26] predates uses of review in CS courses similar to in-flow and multi-stage reviewing. We focus on review scenarios that emphasize using feedback to improve the assignment as in the case of in-flow review, rather than review in general.

In the implementation of a multi-stage compiler, students in Søndergaard's course review one another's work between stages [24]. The evaluation in that work was only in the form of surveys after the assignment, but shows generally positive attitudes from students indicating that they felt the review had helped.

Expertiza [20] allows for multiple rounds of revision and review, but on complete submissions, rather than on intermediate stages of assignments as in Captain Teach. Expertiza also allows students to review one another's reviews, and these meta-reviews are used in assessing grades. Captain Teach also allows students to give feedback on reviews, but we do not use the review process in assigning grades. Expertiza's assignments are generally focused on producing larger, more collaboratively-generated deliverables than in the assignments we study.

Hundhausen, et al. [15] 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.

9.3 Test-focused Assignments

Sauvé and Abath Neto present an assignment structure where students work with acceptance tests from a simulated client in the acceptance test-driven development style (ATTD) [22]. They emphasize the importance of having students review the acceptance tests they are given in order to understand the problem. The tests are written by the instructor rather than by students, so it isn't an instance of peer review. When using this style, they recommend putting intentional errors into the tests to tell if students are paying attention:

> Leave some typos and doubtful points in the scripts on purpose. This will force students to contact you for clarification, since they are not allowed to change the acceptance tests. This also

serves as feedback. If, in 2 weeks' time, no one has contacted you, it is unlikely that any student is making progress in the project.

Buffardi and Edwards study students' testing behaviors under test-driven-development [4, 3]. Students could submit tests and code for automated assessment multiple times before the deadline, receiving in-flow feedback. The point at which a student's tests achieve significant coverage of her code is a key parameter in their analysis. This parameter is much less meaningful in our work, since students submit (and ostensibly write) tests prior to implementations. Our two groups share an interest in leveraging early testing to improve code quality; identifying appropriate roles for peer-review in that process is an interesting open question.

9.4 Peer Testing

A few studies have investigated having students write tests for one another as a form of code review, though they have not reviewed the tests as a standalone artifact.

Smith, et al. have students write tests for one another as part of reviewing [23]. The tests (and any bugs found) are reported to the original authors, who evaluate the feedback, including the tests and overall testing strategy. Smith, et al. do not discuss specific qualities that were evaluated of the tests and reviews, and the testing was more whole-system than in our work. When evaluating reviews, the students could also submit fixes to their program that the review identified; the authors do not provide data on how often or to what degree students exploited this.

Clark studies peer testing in larger software engineering projects across several years [6]. The testing is less automated because it involves using interactive interfaces. Afterwards, student programmers are asked to evaluate their testers with a rubric that measures helpfulness.

Reily, et al. have students submit test cases as part of a peer review process [21], along with Likert and open-response questions. They do not focus on reviewing test suites themselves, however, instead using tests as a kind of concrete feedback on implementations, as part of a rubric.

Gaspar, et al. surveyed students on their perceptions of Peer Testing, in which students share their test suites with classmates [13]. The time at which their students submitted tests relative to the due date is not clear, though the survey asks students about the impact of Peer Testing on their programs. Students generally perceived benefits to trading test suites. Students did not provide feedback on each others' tests, so this use of Peer Testing captures only one of the reviewing roles required of Captain Teach students.

Buffardi and Edwards study automated, rather than peer-written, feedback on test cases [4]. The feedback was different in two main ways: the style was test-driven, with students encouraged to maintain complete code coverage as much as possible, and the feedback took the form of hints on students' solutions, rather than tests, and the feedback was *earned* by submitting good tests. We evaluate tests differently, relative to the problem via thoroughness and correctness, so we don't have a direct analog for the metric of code coverage. 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 [16]. 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.

Appendix A – Coding Rubrics

Review Feedback, 2013

Used for coding review feedback as summarized in section 4.5.3.

```
Fine to read the review contents if necessary to interpret a feedback comment.

Courtesy: inconsequential short phrase, no value judgement
 ** If mark this, mark nothing else
 - "thanks", "cool", ":-)"

Positive Reinforcement: generic but positive comment
  ** Use for any expression of thanks in conjunction with another marking
  - includes :-)
  - "good suggestion"/"good catch"/"this will be helpful"

explain behavior: personal comments such as coding style, work habits
  - "stopped running tests cases"
  - "allergic to long names"

explain code: clarifies/justifies code referenced in review
  - "this is tested elsewhere"

Constructive: some specific feedback about review contents
  - "first comment isn't clear"
  - "point to a specific test case"
  - "you aren't giving justification"

Nothing constructive: blanket negative statement about review
  - "this is useless"
```

```
Confused About Review: expresses not understanding some comment
  - "first comment isn't clear"
  - "which test case are you referencing?"

Acknowledge Error: author acknowledges having made a mistake
  - "I missed X"
  - "Yeah, I fixed that"
  - "I changed that"
  - a simple "I added that" does not count, no clear ack there

Specific Dispute: Disagrees with specific claim from review
  - "I did have the right behavior"
  - "I did include X"
  - "... but I did X"

Defensive Tone: writing suggests author feeling defensive
  - "I knew how to do that"

Sarcastic or Rude Tone:
  ** sarcasm should be directed to/about reviewer, not CT
  - "Are you actually paying attention"
  - "Can't you speak English"

Criticize Reviewer: comment directed personally at reviewer
  - "you aren't reading carefully"
  - "your English isn't good"

Discuss Assignment: question or discussion about problem statement
  - "no indication problem requires"
  - "I'm not sure what to do with X"

Will Act: Intent to make a specific edit/change
  ** tense (future/past) isn't important, but comment should suggest
  **   that review had something to do with the fix getting made
  **   (so something like "done" doesn't count here)
  - "I'll do that"
  - "I'll test the base case"

Caught On Own: comment suggests that author found reported problem on own
```

```
  - "I figured that out when I wrote the code"
  - "yeah, I noticed that later"

Meta-learning: Insight gained about testing/coding/how to work
  ** Expectation that "learning" would apply more broadly than this assignment
  - "I should have tested using more operators"

Process helped: noted insight resulting from review process
  - "saw this after reading others' reviews"

CT Interface: comment about CT as a tool
```

Review Content, 2014

This rubric was used to code the reviews as summarized in section 6.3.3.

```
- Uncodable

  For whatever reason, this fits into none of the categories below.  It could
  be unintelligible writing, completely off-topic, or have interesting content
  that doesn't fit anywhere.

- No Text/Empty

  Is somehow missing content that would be needed to code (easy to tell
  programmatically if there are likert responses or not, so no need to code "no
  text" and "empty review" differently)

- Codable

  Complimentary/Positive

    - 0 - Neutral or negative tone throughout, includes just stating facts with
          no value judgment, like "tests update, init, and movement operations".
          Includes short, essentially content-less comments like "looks good",
          "looks reasonable", "all are correct"
    - 1 - Positive tone with some kind of minor comment like "good job covering
          corner cases," "solid nesting," "covers a good range of inputs".
```

- 2 - Very specific or excited comment, like "wow that's an amazing amount of detail" or "i like the way you combined map and sort"

A C/P rating of 0 or 1 with no other boxes checked is defined to be a "LGTM" review.

Take the MAX C/P rating across the whole review. This counts even in the cases where students were explicitly prompted to say something nice, as long as they said something specific enough. If they are asked to say something nice and just say "good corner case coverage", then that is still just a 1.

- Hostile/Negative

Outright pejorative wording like "this test suite is bad", or referring to the submitter rather than the code "you didn't do a very good job," "you didn't try very hard"

Just pointing out mistakes doesn't make a review hostile or negative, rather we're looking for negative value judgements about the work or the author. However, pointing out test case errors can be overly negative by being too vague and unhelpful -- e.g. "none of these tests are right". To qualify as a dispute of behavior, the comment needs to at least single out a particular test or discuss the input or output.

A review can be hostile/negative and still have any C/P rating, by having those features in different parts of the review.

- Discusses Assignment/Desired Program Behavior/Problem Spec

Refers explicitly to the assignment writeup, or discusses the specification of the program. May discuss it without clearly pointing out a mistake, or may be part of disputing correctness of a test case. For example "assignment said to assume pre-sorted lists," "the boards will always be full when this is called."

Discussing the spec may go along with pointing out a mistake, but not necessarily.

- Identifies (Potential) Mistake

  Points out a programming or testing error. This includes errors in the implementation language (e.g. Pyret doesn't have .length on strings, that function takes two arguments and you gave it one), and also includes mismatched inputs and outputs in tests, testing errors, wrong answers, etc.

  Note that it would be nice to tell apart programming mistakes from testing misunderstandings, but this is way too hard to do without training all the coders on all the assignments. Some mistakes may go along with discussing the assignment/problem spec.

- Suggests Additional Test(s)/Identifies Missing Tests/Untested Feature

  This includes both negative statements like "This doesn't test the binary operators" or "No error cases" as well as positive ones like "You should test collections with more than a few elements". The goal is to point out a way the submitter should test more.

  There should be some semblance of an idea of *what* test(s) should be added, also. For example, just saying "there aren't enough tests" doesn't count. The comment should at least mention something problem-related to test (e.g. at least mention "the empty list", "test nested expressions", "test addition", etc). Just saying "needs many more tests" doesn't count.

- Coding tip/style comment/code or tests structure comment

  These are often things about comments and structure, like "did you know you can highlight all and press tab to reindent," or "you can name your check blocks and it shows up in the output," or "you should break this up into more check blocks"

  These are tips that ought to apply across assignments, and wouldn't affect the correctness or completeness of running the test suite.

  This covers some of what used to be plan/strategy discussion, including comments like "creating a new collection every time you do a simple test is inefficient -- you could re-use the same one".

- Reviewer Gained/Confirmed Understanding

  For cases where the reviewer says "This one does what mine did, so we agree!", and also if the reviewer says "Oh I see a much better way to organize my tests now, cool!"

  The reviewer needs to explicitly indicate their agreement or what they learned, can't infer that they learned because they think something is "clever".

Review Reflection, 2014

Process Comment (Frustration or Otherwise):

General category for students who had something technical or unfortunate happen in the review process, didn't submit on time, didn't get reviews submitted, etc.

'I didn't get any reviews until right before the deadline';

'I got empty submissions to review'

No Text

Use this if the prompt appears, but there is no text (this is distinct from the case where the prompt does not appear, and that difference is easy to tell programmatically).

No Changes to Submission

For when students explicitly say 'none', 'no changes', 'N/A' etc. Make sure to check this if they indicate things were unchanged.

If they say something about their own learning, but don't say anything about an actual change to their submission, DO check this category.

If they say something like 'nothing really, but I did add some tests', DO NOT check this, they negated it by saying they actually did something.

Fixed/found Bug in Tests

Includes cases where students acknowledge an error -- 'pointed out that I had been calculating vectors wrong all along', even if the student doesn't explicitly say 'and I fixed it' Also includes if a test is removed for being incorrect.

Added or Identified Missing Test(s)

For when students add new specific tests or new categories of tests, for example 'I added short-circuiting, ordering, and more nested cases.'

We ASSUME (based on the way the prompt is written) that if a student simply lists a feature, it means they added tests for that feature, so if the student simply writes 'nested lambda,' it means they added tests.

Changed Implementation/Code

If the student refers to changing their implementation (not just their tests), use this. Generally this comes up when a student refers to function names (e.g. 'I edited is-valid to have more checks'), rather than adding new tests.

Enhanced Understanding/Reflected on Behavior

When the student learns something about the assignment or in general. This could be about the language they are using, about the specific problem they are solving.

Often goes along with fixed/found bug or added test, for when students report that they understand part of the problem better. The vectors comment in the fixed/found bugs would could for this category as well.

This combines the previous categories of reflected on own behavior and enhanced understanding.

Unhelpful/incorrect/disagree

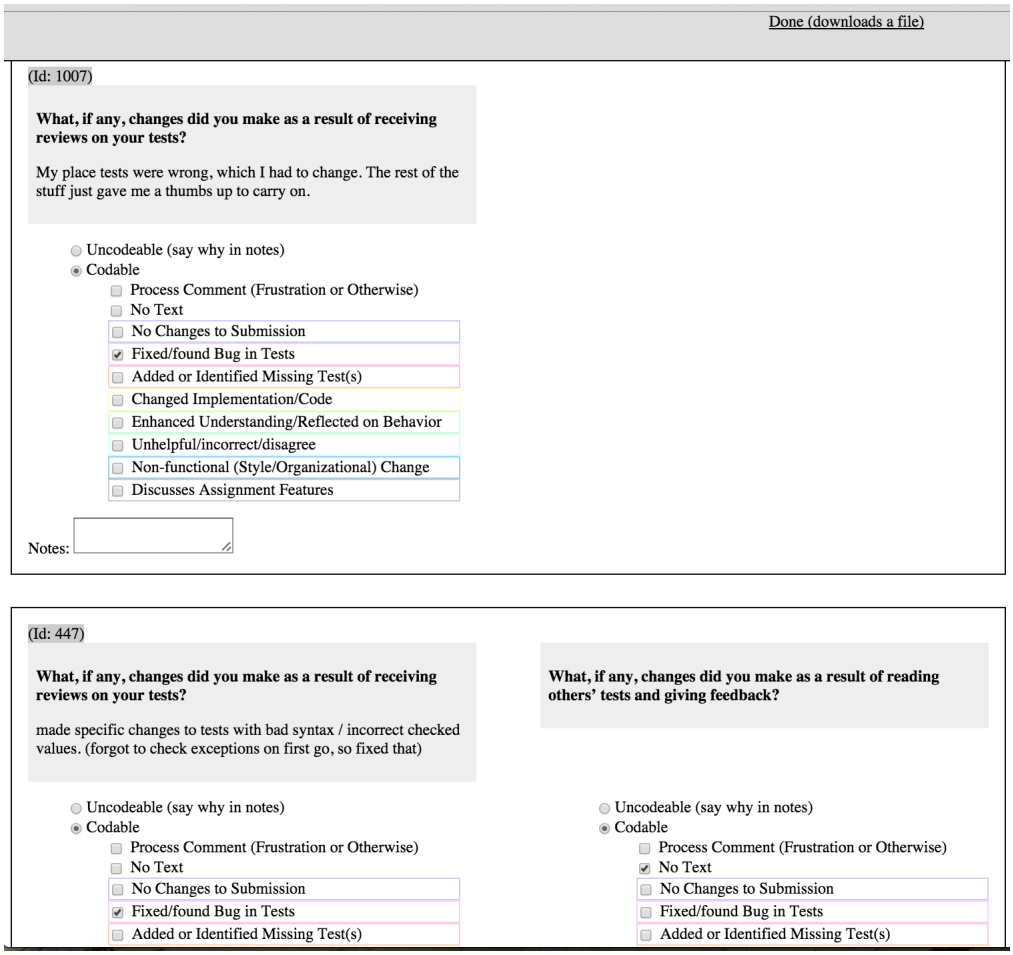
'The reviewers just OK'd my work and moved on', 'Not much useful feedback in reviews', 'didn't see anything new'. Includes indications that the review is incorrect, or that the reviewee disagrees with the feedaback they received.

Non-functional (Style/Organizational) Change

For when students fix comments, reorganize code, use better programming patterns, etc, as a result of seeing others' code or receiving reviews. 'I changed my tests to use multi-line strings'. This should indicate a concrete change to their tests.

Discusses Assignment Features

For when students explicitly reference features of the assignment. For example, they may refer to particular data structures ('the Sudoku board') or features, like laziness or records.



The coding manual text for each category appeared when the cursor hovered over a checkbox option. A similar interface (though without category descriptions on hover) was used to code review content.

Figure 9.1: A screenshot of the interface used to code review reflection

Appendix B – Review Surveys

These are the surveys whose responses are summarized in section 5.3.

CSPL Review Survey, 2014

Edit this form

Review of Sweeps Survey

Since we switched the style of initial test submission for review, we're curious if you like it better or not, and how it interacts with your feelings on review. This survey is anonymous and doesn't affect your grade.

By the "sweep", we mean the initial 5-10 tests you are now submitting for review in lieu of a large, comprehensive suite as in the first few assignments.

* Required

Since switching to the sweep (the new style of test submission), review has been helpful when I've participated, whether receiving or writing reviews. *

I would prefer to be included in review in some form on every assignment using the sweep. *

I would prefer to both give and receive reviews on assignments using the sweep. *

I would prefer to always only give reviews on assignments using the sweep. *

I would prefer to always only receive reviews on assignments using the sweep. *

Submit

CS1.5 Review Survey, 2014

Edit this form

Review of Sweeps Survey

This survey is anonymous and doesn't affect your grade.

* Required

I would prefer to be included in test review in some form on every assignment. *

I would prefer to both give and receive test reviews on every assignment. *

I would prefer to always only give test reviews on each assignment. *

I would prefer to always only receive test reviews on every assignment. *

Submit

Powered by

This content is neither created nor endorsed by Google.

Report Abuse - Terms of Service - Additional Terms

CSPL Review Survey, 2013

Part 2 of the survey contained questions that were more typical course evaluation questions. These are the questions reported on in chapter 4.

For each listed assigment, indicate if the *reviews you received* were helpful in improving your assignment. *

	Reviews were not helpful	Reviews were somewhat helpful	Reviews were very helpful
interp-basic	○	○	○
calculate-locals	○	○	○
interp-state	○	○	○
interp-records	○	○	○
interp-with	○	○	○
calculate-locals-with	○	○	○
interp-lifting-locals	○	○	○
calculate-locals-typed-with	○	○	○
written-scope	○	○	○
written-control	○	○	○
written-types	○	○	○
written-gc	○	○	○
written-lazy	○	○	○

For each listed assigment, indicate if *reviewing others' work* was helpful in improving your assignment. *

	Reviewing others' work was not helpful	Reviewing others' work was somewhat helpful	Reviewing others' work was very helpful
interp-basic	○	○	○
calculate-locals	○	○	○
interp-state	○	○	○
interp-records	○	○	○
interp-with	○	○	○
calculate-locals-with	○	○	○
interp-lifting-locals	○	○	○
calculate-locals-typed-with	○	○	○
written-scope	○	○	○

written-control	○	○	○
written-types	○	○	○
written-gc	○	○	○
written-lazy	○	○	○

For each listed assigment, indicate whether the review process for tests or code was more helpful. Consider both seeing others' work and the reviews you received. *

	Test reviewing was more helpful	Code reviewing was more helpful	They were equally helpful
interp-basic	○	○	○
calculate-locals	○	○	○
interp-state	○	○	○
interp-records	○	○	○
interp-with	○	○	○
calculate-locals-with	○	○	○
interp-lifting-locals	○	○	○
calculate-locals-typed-with	○	○	○

Part 1 of 2 complete

Please go on to part 2 at https://docs.google.com/a/brown.edu/forms/d/1eVQlqHmdaciepje_imJ0OS46c2QMiqzzQ6g-Ewlf9eE/viewform.

It may be convenient for you to open this link in a new tab before submitting, since it won't be visible after you submit.

☐ Send me a copy of my responses.

Submit

Never submit passwords through Google Forms.

CS1.5 Review Survey, 2013

Part 2 of the survey contained questions that were more typical course evaluation questions. These are the questions reported on in chapter 4.

For each listed assigment, indicate if the *reviews you received* were helpful for improving your assignment. *

	Reviews were not helpful	Reviews were somewhat helpful	Reviews were very helpful
Nile	○	○	○
Oracle	○	○	○
Updater	○	○	○
Brown Heaps	○	○	○
DocDiff	○	○	○

For each listed assigment, indicate if *reviewing others' work* was helpful in improving your assignment. *

	Reviewing others' work was not helpful	Reviewing others' work was somewhat helpful	Reviewing others' work was very helpful
Nile	○	○	○
Oracle	○	○	○
Updater	○	○	○
Brown Heaps	○	○	○
DocDiff	○	○	○

For each listed assigment, indicate whether the review process for tests or code was more helpful. Consider both seeing others' work and the reviews you received. *

	Test reviewing was more helpful	Code reviewing was more helpful	They were equally helpful
Nile	○	○	○
Oracle	○	○	○
Updater	○	○	○
Brown Heaps	○	○	○
DocDiff	○	○	○

Part 1 of 2 complete

Please go on to part 2 at

Appendix C – Retrospective Survey

This survey is reported on in section 8.1.

Edit this form

Peer Review of Tests Survey

If you're receiving this survey, you used Captain Teach for peer review of tests in one or more of 2013, 2014, and 2015.

Fall 2013

In which class did you use Captain Teach in fall 2013?

- ◯ CS019
- ◯ CS173
- ◯ I didn't use it in 2013 (skip this section)

The peer review process we used in fall 2013 was useful.

- ◯ Strongly agree
- ◯ Somewhat agree
- ◯ Neutral
- ◯ Somewhat disagree
- ◯ Strongly disagree

Looking back, has your feeling of the process's usefulness changed from how you felt when you participated in fall 2013?

- ◯ I now feel it is much more useful
- ◯ I now feel it is somewhat more useful
- ◯ No change
- ◯ I now feel it is somewhat less useful
- ◯ I now feel it is much less useful

Fall 2014

In which class did you use Captain Teach in fall 2014?

- ◯ CS019
- ◯ CS173
- ◯ I didn't use it in 2014 (skip this section)

The peer review process we used in fall 2014 was useful.

- ◯ Strongly agree

- ◯ Somewhat agree
- ◯ Neutral
- ◯ Somewhat disagree
- ◯ Strongly disagree

Looking back, has your feeling of the process's usefulness changed from how you felt when you participated in fall 2014?

- ◯ I now feel it is much more useful
- ◯ I now feel it is somewhat more useful
- ◯ No change
- ◯ I now feel it is somewhat less useful
- ◯ I now feel it is much less useful

In fall 2014, how often did you opt to participate in review when given the choice?

- ◯ I always participated in review
- ◯ I participated most of the time
- ◯ I participated a few times
- ◯ I always opted out of review

Fall 2015

In which class did you use Captain Teach in fall 2015?

- ◯ CS019
- ◯ CS173
- ◯ I didn't use it in 2015 (skip this section)

The peer review process we used in fall 2015 was useful.

- ◯ Strongly Agree
- ◯ Somewhat Agree
- ◯ Neutral
- ◯ Somewhat disagree
- ◯ Strongly Disagree

Looking back, has your feeling of the process's usefulness changed from how you felt when you participated in fall 2015?

- ◯ I now feel it is much more useful
- ◯ I now feel it is somewhat more useful
- ◯ No change
- ◯ I now feel it is somewhat less useful

- ◯ I now feel it is much less useful

In fall 2015, how often did you opt to participate in review when given the choice?

- ◯ I always participated in review
- ◯ I participated most of the time
- ◯ I participated a few times
- ◯ I always opted out of review

General

Of the courses you've taken since, in which would test review have been helpful (if any)?
Leave blank if test review wouldn't have been helpful in other courses you've taken.

Have you used concepts or skills you learned from peer review of tests in other contexts?
Leave blank if you haven't re-used these concepts or skills in other contexts

- ☐ Yes, in other coursework
- ☐ Yes, in internships
- ☐ Yes, in research programming
- ☐ Yes, in personal projects
- ☐ Other:

Use this space to elaborate on any of your responses above, or give other open ended feedback:

Submit

Never submit passwords through Google Forms.

Bibliography

[1] Alberto Bacchelli and Christian Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. International Conference on Software Engineering*, 2013.

[2] Kent Beck. Extreme Programming Explained: Embrace Change. Addison Wesley, 2000.

[3] Kevin Buffardi and Stephen H. Edwards. Effective and Ineffective Software Testing Behaviors by Novice Programmers. In *Proc. International Computing Education Research*, 2013.

[4] 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.

[5] O. L. Chapman. The White Paper: A Description of CPR. 2001. `http://cpr.molsci.ucla.edu/cpr/resources/documents/misc/CPR_White_Paper.pdf`

[6] Nicole Clark. Peer testing in software engineering projects. In *Proc. Australasian Computing Education Conference*, 2004.

[7] 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.

[8] Jason Cohen, Steven Teleki, and Eric Brown. Best Kept Secrets of Peer Code Review. SmartBear Software, 2013.

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

[10] 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.

[11] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, pp. 182–211, 1976.

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

[13] Allessio 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.

[14] 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.

[15] Christopher D. Hundhausen, Anukrati Agrawal, and Pawan Agarwal. Talking About Code: Integrating Pedagogical Code Reviews into Early Computing Courses. *ACM Transactions on Computing Education* 13(3), 2013.

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

[17] Joe Gibbs Politz, Joseph M. Collard, Arjun Guha, Shriram Krishnamurthi, and Kathi Fisler. The Sweep: Essential Examples for In-flow Peer Review. In *Proc. Special Interest Group on Computer Science Education*, 2016.

[18] Joe Gibbs Politz, Shriram Krishnamurthi, and Kathi Fisler. In-flow Peer Review of Tests in Test-First Programming. In *Proc. Innovation and Technology in Computer Science Education*, 2014.

[19] Joe Gibbs Politz, Daniel Patterson, Shriram Krishnamurthi, and Kathi Fisler. CaptainTeach: Multi-Stage, In-Flow Peer Review for Programming Assignments. In *Proc. Innovation and Technology in Computer Science Education*, 2014.

[20] Lakshmi Ramachandran and Edward F. Gehringer. Reusable learning objects through peer review: The Expertiza approach. In *Proc. Innovate: Journal of Online Education*, 2007.

[21] K. Reily, P.L. Finnerty, and L. 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.

[22] Jacques Philippe Sauvé and Osório Lopes Abath Neto. Teaching software development with ATDD and easyaccept. In *Proc. Special Interest Group on Computer Science Education*, 2008.

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

[24] Harald Søndergaard. Learning from and with Peers: The Different Roles of Student Peer Reviewing. In *Proc. Innovation and Technology in Computer Science Education*, 2009.

[25] The College Board. AP Computer Science Principles, Draft Curriculum Framework. 2014.

[26] Keith Topping. Peer Assessment Between Students in Colleges and Universities. *Review of Educational Research* 68(3), pp. 249–276, 1998.

[27] Lawrence G. Votta Jr. Does every inspection need a meeting? In *Proc. Foundations of Software Engineering*, 1993.