# In-Flow Peer Review

Dave Clarke
Uppsala Universitet
dave.clarke@it.uu.se

Tony Clear
Auckland University of Technology
tony.clear@aut.ac.nz

Kathi Fisler
WPI
kfisler@cs.wpi.edu

Matthias Hauswirth
University of Lugano
Matthias.Hauswirth@usi.ch

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

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

Ville Tirronen
University of Jyväskylä
ville.e.t.tirronen@jyu.fi

Tobias Wrigstad
Uppsala Unviersitet
tobias.wrigstad@it.uu.se

## Abstract

Peer-review is a valuable tool that helps both the reviewee, who receives feedback about their work, and the reviewer, who sees different potential solutions and improves their ability to critique work. *In-flow* peer-review (IFPR) is peer-review done while an assignment is in progress. Peer-review done during this time is likely to result in greater motivation for both reviewer and reviewee. This working-group report summarizes IFPR and discusses numerous dimensions of the process, each of which alleviates some problems while raising associated concerns.

## 1. In-Flow Peer-Reviewing

Peer-review has been employed for various reasons in Computer Science courses [54]. It is a mechanism for having students read each others' work, learn how to give feedback, and even to help with assessment. Indeed, of the six major computational thinking skills listed in the current draft of the AP Computer Science Principles curriculum [12], the fourth is:

> 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.

Peer review has uses beyond merely evaluating programs. Writing benefits from peer review, as do other artifacts that aren't just programs, like design diagrams, test suites, formal models, documentation, and presentations. All of these artifacts are also fair game for peer review in computer science courses and more, and peer review addresses similar underlying learning goals of evaluation and explanation of existing work.

This working group explored a particular variant of peer-review called *in-flow peer review* [22]. In this model, peer review occurs while an assignment is in progress, before students submit their work for final grading. Performing peer-review in-flow has several potential benefits:

- It helps students better understand the problem specification. If the work they see others doing is inconsistent with their understanding, one or the other (or both!) might be confused. It is better to discover this while the assignment is in progress than after it is over.
- Students are motivated to read feedback they get since it can affect their performance on the current assignment. In contrast, feedback given when the assignment is over may get less attention if students have moved on to other assignments.
- Students can apply what they learn from seeing examples of one another's work, and also learn to exercise judgment when evaluating existing solutions. When a student sees another's

work, they do not know the quality of the work they see: it could be better than their own work, but it could also be worse. This takes some potential problems with plagiarism and turns them into a part of the learning process.

- It further emphasizes the *comparative* examination of work against a student's own.
- It exposes students to a standard component of industrial software development.

Several challenges arise with this model, including figuring out how to decompose assignments for meaningful reviews, how to prevent students from gaming the process to avoid doing their own work, how to minimize the extra time this takes to complete homeworks, and how to help students not be led astray by weak or inaccurate reviews. Considering the potential learning objectives of IFPR, rather than viewing it merely as a way to scale grading (a use of peer-review in MOOCs [33]), IFPR adds subtlety to all of these questions. Liu and Carless distinguish between *peer feedback* and *peer assessment* [34], where the latter's goal is grading, and the former's is collaboration – IFPR is primarily focused on peer feedback, not peer assessment.

This report summarizes activities of a working group around the promises and pitfalls of in-flow peer-review in computer science classes. The group members represented several countries and taught various courses at various levels (though the majority taught courses related to programming, programming languages, or various aspects of software development). We arrived at several different learning objectives an instructor might have for using peer review in a course (section 4) that drove our discussion. In addition, prior to the group's in-person meeting, each group member created two assignments for in-flow peer-review. These case studies, which are summarized in two figures (figure 2, figure 3), also helped to form the basis of many of our discussions.

## 2. An IFPR Roadmap

IFPR is a mechanism open to many policies. These policies are a function of a course's goals, student maturity, cultural context, and more. Therefore, an instructor who chooses to use IFPR will have to make several decisions about exactly what form they will employ. This section briefly outlines some of these decision points, with references to the rest of the document for more details.

### 2.1 The IFPR Process

IFPR follows a particular process for assignments. In order to have an in-flow component, the assignment requires at least one reviewable submission that occurs before the final deadline of the assignment. This requires thinking through a few procedural questions that all in-flow assignments must address:

- The choice of submissions. How should an assignment be broken down into multiple stages? Even in a programming assignment, there are many choices: tests before code; data structures before tests; design documents and architectural specifications before code; multiple iterations of code; and so on. Some choices raise more concerns regarding plagiarism, while others only work under certain assumptions about software development methodologies. (Section 6.1)
- The distribution of reviewing. Should reviewing be distributed in order of submission? Randomly? Between students of similar or opposite attainment levels? Synchronizing review across students enables more policies on reviewer assignments, but incurs overhead for students and staff through more course deadlines. (Section 6.2)
- The manner in which reviews are conducted. This includes the choice of review "technology": should reviewing by mediated by a computer application or should it be done face-to-face (perhaps as a small group meeting around a table)? This also includes the use of rubrics: On the one hand, rubrics for reviewing guide the reviewer and may result in more concrete, actionable outcomes. On the other hand, a rubric can result in less constructive engagement and may result in important issues being missed. (Section 6.3.1)

### 2.2 Issues Surrounding IFPR

There are a number of other cross-cutting issues that inform the choices made in the in-flow process, and affect the appropriateness and effectiveness of IFPR in particular contexts:

- The role of anonymity. When, if ever, should authors and reviewers know about each others' identity? Using single- and double-blind reviewing systems introduces trade-offs between protecting students' identity, creating the potential for abuse, and introducing them to norms of professional behavior, all of which need to be taken into account. Anonymity may enable more students to participate comfortably, at the cost of missed opportunities for creating cultures of collaboration and professional working behavior. (Section 7.3)
- The role of experts. Experts can play any number of roles from being entirely hands-off and treating this as an entirely student-run process (presumably after introducing the process and its purpose) to intervening only periodically to constantly monitoring and even grading the responses. These roles set different tones between students regarding expertise and authority, but also help set standards while students are new to the process. (Section 7.4)
- Suitability for non-majors. While IFPR is easy to justify for majors because of its correspondence to industrial practice (code-reviews), the industrial argument makes less sense for non-majors. Objectives around collaboration and creating standards of evaluation, however, seem to apply to both majors and non-majors. (Section 7.5)
- The relationship to pair-programming. Pair-programming might appear to also be a kind of IFPR—indeed, an extreme version of it. However, the two processes are somewhat different and complementary. (Section 8.1)

## 3. Terminology

Throughout this report, we use the following terminology for the various artifacts, roles, and aspects of IFPR (figure 1 illustrates these graphically):

- An *exercise* is a problem set or assignment associated with a course; it may consist of multiple independent subproblems.
- A *stage* is a problem or task within an exercise that will be sent out for peer review.
- A piece of work on which a student will be reviewed is called a *submission*: this could be a piece of code, a paper, a presentation, or any other work on which peers will provide feedback.
- The *author* of a submission is the student who did the work associated with the submission (and who presumably would receive any grade associated with the submission, as well as any reviews about the submission).
- A *review* is written by a *reviewer* and responds to a specific submission.

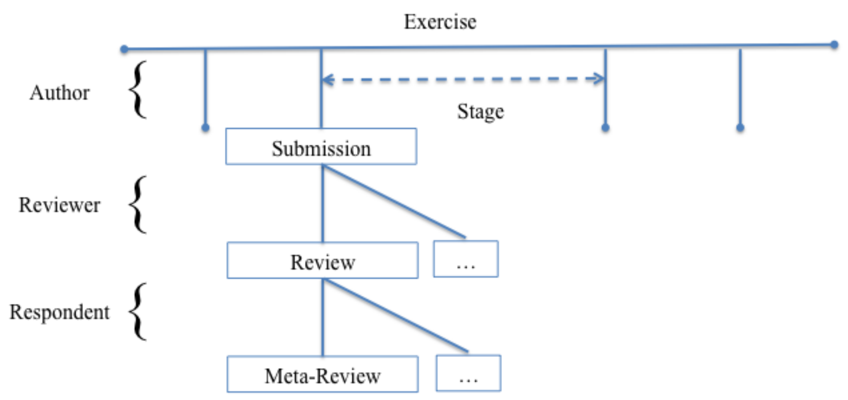

Figure 1: Illustration of Key Terminology

- A *meta-review* contains feedback on a review: this could be a grade of a review produced by course staff, feedback from the original submission author back to the reviewer, or any other commentary on the contents of a review. The provider of a meta-review is called a *respondent*.
- A *reviewing assignment* indicates which reviewers are expected to produce reviews for each submission.

## 4. Educational Goals of IFPR

Both peer review and the in-flow variant target a complex and interesting set of educational goals, some student focused and some instructor focused. Working group members were surprised at the subtleties that these goals brought to questions about how to configure IFPR. Indeed, many found discussions of the educational goals (and their impacts) the most thought-provoking aspect of our discussions. We lay out the goals here, referring back to them as we discuss configurations of IFPR throughout the report.

### 4.1 Student Learning Objectives

Fundamentally, IFPR fosters collaborative learning in which students can practice several critical skills:

- Assessing whether another's work satisfies problem requirements
- Providing actionable, useful, and appropriate feedback to others
- Extracting high-level design choices from another's work
- Comparing others' high-level design choices and practices to one's own
- Deciding whether to adopt or ignore particular feedback or ideas
- Learning to value and grant authority to feedback from peers

The first two items arise primarily in students' role as reviewers, and are common to all forms of peer-review. The remaining tasks arise more in students' roles as recipients of reviews, and have more urgency in an in-flow context. In the context of Bloom's taxonomy [6], these skills move students beyond "remember," "understand," and "apply" into "evaluate". They engage students in reflection and meta-cognitive thinking about their own work, while also requiring students to be able to communicate technical issues clearly to others.

Beyond these goals, regular comparison of one's own work to that of others can help students calibrate their abilities. In particular, it should provide means for students to gain confidence and self-efficacy in their work, and in discussing the works of others.

The extent to which IFPR targets these goals depends significantly on the artifacts students are asked to review, the criteria or rubrics through which they produce reviews, the means through which students are expected to respond to reviews, and the feedback students receive on their reviews. Some configurations of peer review, particularly those designed simply to scale grading, naturally and necessarily de-emphasize some of these goals. Section 7 explores these tradeoffs in detail.

The emphasis on collaboration in these goals illustrates that IFPR is an example of a Contributing Student Pedagogy (CSP), a pedagogy in which students (1) contribute to the learning of others and (2) value the contributions of other students. A 2008 ITiCSE working group report describes various facets of such pedagogies [24]. IFPR targets the second criterion (valuing the contributions of other students) more than traditional, post-submission peer review. Various parameters in implementations of IFPR affect the extent to which students contribute to the learning of others in practice: shallow reviews, for example, arguably meet the letter but not the intent of a CSP. Separately, IFPR has goals beyond CSP: writing reviews offers benefits to the reviewer as much as the reviewee, and often the learning goals that inspire IFPR (and peer-review in general) are more focused on the reviewer than the reviewee. Nonetheless, many of the theoretical underpinnings of CSPs also apply to IFPR, and thus affect the ideas in this report.

Of course, IFPR also has the potential to interfere with student learning. Reviewing asks students to switch between very different tasks (programming and reviewing); depending on the timing of reviewing, this could affect students' cognitive load. Careful design of exercises is important: allowing students to share parts of their solutions through reviewing can, for example, affect whether students stay in their zones of proximal development [55].

### 4.2 Instructor Goals

From an instructor's perspective, IFPR can target several objectives, including:

1. Emphasizing the importance of writing in technical contexts
2. Providing human feedback more scalably and more timely than with only expert assessment

3. Providing an additional perspective on how students perceive course material, since students express their understanding in a way other than just their assignment submissions
4. Increasing social interaction within computing and programming, addressing a common misconception about working in the discipline
5. Fostering engagement of and interaction between students from different cultures
6. Helping students improve performance and learning while actively engaged with course material
7. Re-examining plagiarism issues by casting the re-use of classmates' ideas and code in a positive light, and including grading mechanisms that take this into account

The first three objectives arise in most forms of peer-review. The second two arise in general peer-review, though the immediacy of the in-flow context likely enhances their impact. The last two are more directly associated with IFPR.

All of the working group members were interested in IFPR more as a way to enhance students' learning than as a way to scale grading. Indeed, most members were open to (if not already) investing additional staff resources in making sure students were learning reviewing skills from a peer-review process. The members were interested in the insights they could gain as instructors from attending to the third objective (though none believed that grading all of the reviews was scalable or cost-effective).

The group also coalesced around the social benefits of peer review, seeing this as an important aspect of developing competent professionals. Many of our discussions about giving review feedback and whether reviewing should be anonymous revolved around the impacts these issues could have on collaboration and socialization through peer review.

## 5. Examples of IFPR

The case studies from working group members covered a variety of student levels and course types. More interestingly, they varied widely in the kinds of artifacts and processes that they suggested for IFPR. Figure 2 and figure 3 summarize the key parameters of the case studies. The table lists the name and level of the course, describes the assignment in which IFPR was used, describes the submissions that were reviewed, and describes the review criteria for submissions. Assignments with [*] after their descriptions have been used in actual courses; the rest are hypothetical uses proposed by working group members, based on exercises in their current courses.

The source files of the case studies are available at `https://github.com/brownplt/iticse-in-flow-2014/tree/master/in-flow-assignments`, and there are links for each in figure 4 in the appendix.

## 6. The In-Flow Process

Several steps are part of any in-flow assignment, as illustrated in figure 1. This section lays out the process of submissions, reviews, and meta-reviews in more detail, along with the design decisions that the group identified for each activity. Section 7 continues by discussing issues that cross-cut the process.

### 6.1 Stages and Submissions

As the case studies in figure 2 and figure 3 show, IFPR can be used with many different kinds of submissions. Even once an instructor has identified the general class of artifacts to review (such as papers versus code), they can choose different ways to use IFPR to build up to final versions. The group identified four broad choices in the artifacts to be reviewed:

- **Multiple iterations of the same specific deliverable**: This approach is the most similar to existing peer grading approaches, where an entire deliverable is presented for review. This mirrors common practice in courses where students do peer review of written work, which is well-studied in contexts other than computer science [7, 54]. Little extra work in assignment design is required to have students review drafts, so this provides a low-friction way to adapt an existing assignment for IFPR. One consideration is that plagiarism can be more of a problem in programming tasks that are the same across students than in writing tasks where goals are less objective and more variance is expected. We discuss plagiarism (and mitigations) more in section 7.1.

  Case studies 6, 9, and 12 included submission steps that were prototypes or drafts of the final product.

- **Multiple iterations of an evolving deliverable**: Some projects don't have drafts so much as an evolving set of specifications and deliverables. For example, in a long-term software engineering project, the demands of the system may change over time as new requirements are discovered. This is distinct from multiple iterations of the same deliverable because the goal itself changes along with the submission. Both high-level feedback on the direction of the project and low-level implementation feedback can be helpful in this setting.

  Case studies 10, 12, 13, and 15 have elements of this approach, where the deliverable's nature changes more over time, and in response to review.

- **Separate deliverables that build on each other**: Often, programming assignments can be broken down into several subproblems, often expressed via decomposition into helper functions or separate classes. If the assignment naturally lends itself to this kind of breakdown, a natural strategy for using in-flow review is to review individual pieces, to catch mistakes in the components before composing them into a final solution. This approach lends itself well to evaluation as well, because the separate components can be assessed in isolation, and on the initial, pre-review submission as well as after final submission. Also, if the instructor (rather than the students) decides on the decomposition, they have a lot of control over the path students take through the assignment, which can inform decisions about rubrics and feedback guidance.

  Case studies 1, 2, 5, 7, 8, and 15 use this approach.

- **Incremental views of the same deliverable**: Programming problems can have a number of distinct artifacts associated with them beyond code: They have documentation, tests, specifications, underlying data representations, measurable behavior like time and memory complexity, visualizations, and more. Having students produce different views on the same problem, with review in between, is another way to break up the assignment. For example, an assignment might proceed with a test-first or test-last approach, with review before or after a test suite is written. This focuses students on different aspects of the problem in their different stages of review.

  Case studies 3, 6, 10, 13, 14, and 15 take this approach, where submissions are different kinds of artifacts, but all contribute to the understanding of a single artifact.

The nature of the reviewed submission naturally affects the time required for review, as well as the amount of expert guidance required (students will have more experience with evaluating some artifacts over others). In turn, the course's learning objectives

| # | Course and Level | Exercise | Submissions | Review Criteria |
|---|---|---|---|---|
| 1 | Computing for Social Sciences and Humanities (undergrad non-majors) | Cluster data on voting records (US Senate) to identify senators with similar ideology | Code and tests for instructor-defined subsets of overall functionality | Provide scores from 0 to 100 on each of (a) whether tests meaningfully capture the assignment purpose, and (b) whether code performs the corresponding computation correctly |
| 2 | CS1 | Write code and assertions for various components of a pinball game | Work so far on subset of functions designated by instructor | Rate readability and correctness; additional free-form comments |
| 3 | Advanced CS1 with Data Structures | Design a data structure for incremental and functional updates on trees [*] | Datatype definition with instances of the data, test cases, complete programs | Indicate whether (a) data structure can support required operations within time bounds, (b) interesting examples of data are missing, (c) tests offer good coverage and are correct |
| 4 | Programming Fundamentals 2 (2nd semester undergrad) | In-class clicker assignment to explain control-flow through if-statements [*] | CFGs for code snippets (drawn through custom software package) | Provide yes/no assessment of whether CFG is accurate |
| 5 | Imperative and OO Programming Methodology (2nd year) | Implement a program that satisfies a student-selected set of learning goals | Description of learning goals covered by program, program code, and give presentation on how program achieves goals | Instructor-provided template on choice of goals, whether program satisfied them, and presentation quality |
| 6 | Imperative and OO Programming Methodology (2nd year) | Implement simple Pong game in model-view-controller style | Tests, two draft implementations, and a final implementation | questions about whether key components are present; whether tests are reasonably complete and motivated; whether good code practice followed (i.e., naming, structure, identation) |

Figure 2: Summary of case studies: first-year, second-year and non-majors courses ([*] indicates use in an actual course)

should guide the choice of artifacts: preparing students to participate in industrial code review, for example, will be better served by using IFPR on code-based artifacts rather than written papers.

### 6.2 Assigning and Scheduling Reviews

The IFPR process requires both building time into assignments for performing review, and assigning reviewers to artifacts for review.

#### 6.2.1 Scheduling Decisions

The major scheduling decision in IFPR is whether reviewing happens *synchronously* or *asynchronously*. Synchronous reviewing occurs when all students submit work for review (and reviewing commences) at the same time. Asynchronous reviewing occurs when students submit work for review when it is ready, and different students are in different stages of review at the same time. The two scheduling modes have several tradeoffs:

- If all students are forced to submit before reviewing starts, there is the full pool of reviews to draw from in any review assignment strategy (section 6.2.2). In the asynchronous setting, students can only review submissions that occurred before theirs, which can skew the reviewing process if, for example, high-achieving students tend to submit early.
- With intermediate deadlines, all students have the same time to use review information. If there are no intermediate deadlines, students who submit later have less time to use the information from the review process before the deadline.
- In the asynchronous setting, students who want to work at their own pace can, and the process doesn't stop them from continuing with their work. With intermediate deadlines, a student cannot do the work on her own schedule.
- If reviews are available and presented to students after they submit, the problem is more likely to be fresh in their mind. In the synchronous setting, there can be a longer gap between submission and review. It's not clear if one is particularly better than the other: coming back to a problem after not thinking about it for a while can be beneficial, but it also takes time to recall the problem and re-load it into working memory in order to perform review.
- Synchronous reviewing requires extra scheduling overhead that is likely to lengthen assignments for purely logistic reasons. Asynchronous reviewing doesn't require extra scheduling in the assignment, it just changes the kind of work students have to do.

With synchronous reviewing, there is an extra question of whether there should be separate time set aside for reviewing in between submissions (with a review submission deadline), or if reviews can be completely in parallel with the next submission step. This can affect the timeliness of reviews, which can affect how useful the review is to the reviewee as they move forward with the assignment.

#### 6.2.2 Assigning Reviewers to Submissions

Whether reviewing is synchronous or asynchronous, there needs to be a strategy for assigning students to review submissions. As with other design decisions in IFPR, any decision has tradeoffs and depends on course context and goals. We identified both a number of methods for assigning submissions to students to review, and several miscellaneous modifiers that could apply. We note when a particular strategy is more or less appropriate in synchronous or asynchronous settings, as well.

- **Random Assignment**: Perhaps the most obvious and simple method for reviewer assignment is random: each reviewer is assigned one or more submissions at random to review. There are of course many types of randomness; it is probably useful to ensure that all submissions get the same number of reviews,

| # | Course and Level | Exercise | Submissions | Review Criteria |
|---|---|---|---|---|
| 7 | Introduction to Functional Programming (upper undergrad/MS) | Implement Boggle (find all valid words in 4x4 grid) | Decomposition of overall problem into tasks (with QuickCheck assertions), tests, code | check decomposition makes sense, presenting alternative if own differs from reviewed one; try own test suite on the code being reviewed |
| 8 | Advanced Software Design (upper undergrad/MS) | Design and implement subset of a mobile app+server for a game using iterative development | Design documents so far | Free-form comments on comprehensibility, quality of documentation, coverage of use cases, adherence to design principles, and choice of subsystem to implement; concrete examples required to illustrate each point |
| 9 | Collaborative Computing (MS) | Collaboratively produce a research article [*] | Drafts of article | Conference-paper reviewing rubric: questions on suitability for audience, originality and demonstrated knowledge in contribution, evidence for arguments, methods, presentation, etc. |
| 10 | Software Security (upper undergrad/MS) | Find ways to attack a web-based application (black-box, then white-box) | Description of strategy to use in attacking the application in black-box fashion. | Free-form comments on comprehensiveness and appropriateness of attack strategy |
| 11 | Software Modeling and Verification (upper undergrad/MS) | Use model checking to find flaws in a protocol | Proposed model of the system environment and desired properties that should (not) hold under this model | Assess whether model conforms to problem and whether model supports/masks the properties provided with the model; comment on good/bad features of this model |
| 12 | Software Performance (MS) | Develop an extension to the Jikes visual debugger | Proposed extension, prototypes, final artifact | Comment on one thing they particularly like and one aspect that could be improved; evaluate prototypes following in-class presentations by each team; review final artifact for usability, extensibility, and documentation |
| 13 | Logic for System Modelling (upper undergrad/MS) | Write a relational (Alloy) model of an elevator [*] | Model of data components, description of desired properties of model, initial model of elevator operations | Comment on whether components/properties are missing, whether they are reasonable, and whether model is suitably operational or too declarative |
| 14 | Programming Languages (upper undergrad/grad) | Provide a test suite and implementation for a type checker [*] | Tests first, implementations later (submission deadlines not synchronized across students, but must occur in order per student) | Set of $\sim$10 specific questions about test coverage, plus free-form comments on style or organization of test suite; no peer review on implementations |
| 15 | Software Security (MS) | Implement simple on-line web-app on a strict timetable, then create attack trees for it | Initial program, attack trees, and secured application along with review of differences between original and secured application and results of using static analysis and fuzzing tools on the implementation | Free-form comparison to what was done in own solution |

Figure 3: Summary of case studies: upper-level undergraduate and graduate courses ([*] indicates use in an actual course)

for example. In the asynchronous setting, the pool of reviewable submissions from will necessarily be smaller (since only a subset of submissions have already come in); this skews the selection. In an asynchronous setting, random assignment also lacks temporal fairness: the most recent submission isn't guaranteed to be reviewed first, which can weaken the benefit of quick feedback in the asynchronous model.

- **Temporal Assignment**: Reviews can also be assigned in the order submissions were received. It's not clear that this makes much sense for synchronous review, where temporal order is somewhat unrelated to motivations for assigning reviews. However, in the asynchronous case, assigning reviews in the order submissions are received helps ensure that feedback happens as quickly as possible (assuming that students complete reviews at around the same rate). We also note that if students are aware that temporal ordering is occurring, they can collude to time their submissions in order to ensure or avoid particular review pairings. Using a mix of randomness and temporal ordering could alleviate this somewhat, at some minor cost to review turnaround time.
- **By Metric**: There are a number of metrics that could be used to assign reviews with the goal of getting more effective feedback for students. We identified:

  - **Achievement**: Students could be matched to other students with similar or different levels of achievement on past assignments (or, if the assignment can be evaluated automatically, even on the current assignment). Existing research shows that on group work, pairing weak and strong students can help the weak students (though the strong students don't do as well either) [28]. The effects of such an assignment are certainly course- and assignment-specific.
  - **Prior Review Quality**: If the course tracks review quality through meta-reviewing (section 6.4), the system could assign consistently strong reviewers to weak work in order to maximize improvement (again, "weak work" could be predicted by past achievement of students, or by an automatic grading system).
  - **Similarity Between Solutions**: Students may learn more from reviewing a variety of solutions that are different from their own. They may also be more able to review solutions that are similar to their own. Depending on the assignment and learning goals, it could be valuable to groups students based on solution approach or code similarity.

  We expect that by-metric assignments work best synchronously, because it is difficult to perform the assignment until the metric can be measured for all students. Doing a by-metric assignment of reviews asynchronously is possible if the metric is known before submission (e.g. if only using past submission performance), but it would result in some students waiting for their assigned-by-metric reviewee to submit.
- **Student-chosen**: Students could also be involved in choosing which submissions, or other students, they review. For example, a simple model could have all submissions go into a publicly-visible pool of submissions, from which students choose submissions to review. The assignment could require that students perform some number of reviews, and remove submissions from the pool once they have been reviewed enough times in order to avoid a small number of submissions getting more reviews than others. This works with both asynchronous and synchronous-style scheduling, but can be problematic if the limits are too rigid and some students submit very late (leaving no submissions available to review for students who submit early). A solution is to pre-seed the set of submissions with some instructor-provided submissions in order to ensure enough supply.

  Students could also choose partners to review independent of particular submissions. It might be reasonable to switch to a student-chosen partner approach after having assignments with other review assignment strategies, once students have decided they enjoy collaborating. This also works fine in both synchronous and asynchronous styles: students may even arrange to complete their work at a time convenient for their reviewer in order to get the most prompt feedback.

  Papadopoulos et al. explored different strategies for assigning students in peer review in a computer science course on networking [36]. They find that students who select their peers freely perform better (according to experts rating the utility and clarity of reviews) than pairs where the students were assigned (randomly) by the instructors beforehand.

  All student-chosen strategies for review interact heavily with choices about anonymity of the review process, which we discuss in more detail in section 7.3.

In addition to these strategies, there are a few other factors for educators to consider in assigning reviews:

- **Groups vs. individual**: The review assignment strategies in this section aren't limited to only pairs of reviewers and reviewees. It would be perfectly reasonable for a group of reviewers to all create a review together through discussion, whether online or in person. The same parameters of randomness, temporality, and so on apply. Anonymity is possible but more difficult when a discussion between multiple students is involved. We discuss the contents of reviews and review discussions more in section 6.3.
- **Class-wide review**: At the extreme end of group review is a review that involves (potentially) the whole class. This could be, for example, a presentation that the whole class comments on immediately, or a collective review process as in a studio art course, where work is presented and discussed publicly. Using an online tool, work can even be published publicly but anonymously, and allow for any interested class member to comment.
- **Mutual reviews**: Reviewer-reviewee pairs can be mutual or disjoint – students may form pairs (or groups where everyone reviews everyone else) that review one another, or there may be only a one-way connection between reviewer and reviewee. Mutual reviewing could provide more concrete motivation (other than abstract altruism or a grade), if students are helping someone who is actively helping them in return. Mutual reviewing can still retain anonymity.
- **Persistent review assignments**: In an assignment with two or more reviewed stages, reviewers can change at each stage, or continue to be the same throughout the process. One study reports that, for assignments with more than 4-5 stages, switching authors at each stage made the reviewing burden onerous [22]. Continuing to review the same author's work may have lessened this burden, since the comprehension effort from earlier stages would carry over.

Other strategies for assigning reviewers to submissions exist. We have assumed that the reviewers are drawn from within the class (which may not be the case in a peer-mentoring situation [38]). In a class where there are communication or language barriers between students, it may also make sense to assign reviewers so that communication is maximized or the challenge of communicating in another language is maximized. It may also be useful to secretly or not secretly assign instructors or TAs as reviewers and reviewees sometimes in order to guide or monitor the process. We discuss instructor and TA participation in reviews more in section 7.4.

### 6.3 Performing Review

Much of our discussion of how to conduct reviewing focused on *review rubrics*, which can be used to focus student feedback on specific features of the assignment. We considered whether information beyond submissions would help reviewers. We also discussed several forms that reviews could take, noting that technology often guides programming courses towards text-based feedback.

#### 6.3.1 Review Rubrics

Rubrics serve two important goals in any form of peer review: they communicate expectations to reviewers (serving as a form of scaffolding), and they help foster a baseline of quality in all reviews. While these goals suggest highly-structured rubrics, overly structured rubrics can limit reviewers' and authors' attention to the questions on the rubric. They can also provide too much scaffolding, especially once students need to practice evaluating work from scratch. The tradeoffs around designing rubrics must balance these tensions.

Rubric design must consider the rubrics' utility for the reviewer, the author, and the instructor seeking to understand how students are performing. These goals are not necessarily at odds with one another, but may conflict incidentally when picking a particular configuration of rubrics.

The working group identified several potential roles for rubrics:

- **Rubrics as scaffolding for reviews**: Rubrics help students learn how to construct good reviews, especially for students new to the process. A beginning student who is learning to both read and write code might not know where to start in critiquing a program. Prompting with specific questions helps in situations where students don't yet know how to structure a review from scratch.
- **Rubrics for focus**: A rubric can focus students' attention on different questions that reflect the goals of an assignment. For example, it could prompt code-specific questions ("Is this code well-documented?", "Are all the type annotations correct?", etc.), problem-specific questions ("Is there a test for a list with duplicate elements?", "Does this program meet the problem specification for input X?", etc.), or questions that encourage actionable feedback ("Provide a test case that this solution does not pass."). Students may also optionally ask for reviews with a certain focus when offering submissions for review. This might help ensure relevance of the review for the author.
- **Rubrics as an alibi**: Rubrics can be used as alibis for reviewers who fear criticizing works of others, because of cultural values, self-image, or other factors. For example, being asked to point out one part which could be done better, or to identify errors will shift the blame from the reviewer who found the bugs to the instructor who provided the rubric.
- **Rubrics for reviews of good solutions**: In at least one case of using IFPR in the classroom, students reported not knowing what to write when reviewing good solutions [22]. A rubric could explicitly prompt for feedback even on good work (e.g. "What did you like about this submission?", "List one thing you would change, regardless of correctness", "What should the author *not* change in this solution?"), so that students don't simply sign off on a solution as good enough without reflecting and providing some useful feedback.
- **Rubrics for conduct**: Rubrics can guide students towards a professional and appropriate tone for giving feedback, and help frame negative feedback in a constructive way. For example, forcing a review to contain comments on the strengths of the submission under review can soften other criticism. When appropriate, rubrics can guide students towards more constructive language – for example, "This could be done differently" vs. "This is wrong".
- **Rubrics for time management**: Open-ended review tasks don't make it clear how much time students should spend on them. Just having a specific rubric can make it easier for a student to identify when they are done (and estimate the time themselves). A rubric could even specify the amount of time that a student should spend, and how much on each part of the assignment, to ensure that reviewing does not take up more time than intended.

Evolving rubrics across a course or curriculum may offer a good balance between initial scaffolding (for reviewers and authors) and eventual opportunities for both groups to demonstrate critical-thinking skills. One model would evolve rubrics from having fairly targeted questions to asking broad questions: this model gradually removes scaffolding. Another model starts with concrete questions (such as "do these tests look correct") and progresses to questions on more abstract issues (such as "do these tests cover the space of possible inputs") as students master more of the subject material.

A variation on evolving rubrics would allow different students to work with different review forms, depending on their ability as reviewers. This comment arose from the working group members' experience as PC members: members often found overly structured forms to be annoying, feeling they interfered with how they wanted to convey issues with a work. However, they also noted that early on in their paper reviewing career, they appreciated the rubrics' ability to get them past the initial blank form.

A different form of variation might pose more questions to reviewers than are conveyed to authors. This situation could make sense when the review is used to assess the reviewer's understanding of a work, or when too much information in a review might distract the author from the critical information in a review.

Structure enables certain comparisons between reviews. Inexperienced students may benefit from structure when aggregating the feedback of multiple reviews: for example, structure could help students understanding that two or more reviews give contradictory advice. Two students discussing reviews (that they are making or have received) may be similarly helped by an imposed structure. Certain kinds of structure can enable automated analysis of reviews, which can provide useful diagnostics to both instructors and students. Similarly, software tools have the potential to provide richer dashboards when review comments are structured.

Discussing reviews and rubrics with the entire class is another good example of using rubrics for communication. Students or experts might see common problems which should be communicated to all, either by sharing sufficiently general comments with the entire class or even adding an entry to a rubric which brings attention to the issue in subsequent reviews.

#### 6.3.2 Information Provided to Reviewers

In some cases, reviewers can be provided with information beyond the submission. When submissions are source code, for example, reviewers could be given both the submission and information about how the submission held up against an instructor-defined test suite (whether or not that information is available to the submission's author). On the one hand, information such as a test-suite score may reduce the time burden of reviewing; on the other hand, it could have the downside of reviewers only focusing on the issues that auto-grading revealed, masking situations in which the auto-grading missed something important (Politz et al. observed cases in which reviewers were more negative than grades from an instructor-provided test suite [21]).

Additional information for reviewers provides an implicit rubric, subject to the same tradeoffs we discussed regarding rubric structure. Instructors should bear this in mind when considering whether additional information is actually helpful to the overall process.

#### 6.3.3 Forms of Reviews

Reviews can take various forms, from written documents to verbal feedback, from paragraphs to small comments associated with particular fragments of prose or code, and from individual to group-wide feedback. For written reviews, the group noted the general applicability of plain text, but noted that modern software tools (such as Github and other graphical version-control tools) enable targeted comments and conversations between authors and reviewers, down to the line number in a particular revision. These conversations have more more structure than untargeted comments about the entire submission.

In some situations, non-text artifacts can be effective, just as not all submissions need to be code. Code architecture diagrams can be critiqued and marked up with freehand annotations, pictures of the state of running program can be drawn, and even code patches can be an effective way to convey comments. This is one case in which the default technology that authors of tools for in-flow peer review might choose could be limiting.

The Informa tool allows students to give live feedback on problems with several interfaces that could also be useful for review [26]. For example, during a Java program comprehension task, students use a drawing tool to create a graphical representation of the heap at particular program points. Another example had students highlight portions of code that exhibited certain behavior or had a certain feature. Both of these interfaces go beyond simple text or scalar feedback, and can be used to provide richer information in reviews.

Face-to-face reviews are another form, whether solely between students or moderated by TAs or instructors. Moderation can make arguments more constructive, guide discussion towards relevant points, and make a face-to-face meeting less intimidating. Moderated review moves the process more towards a studio-like setting, and may be appropriate especially for teaching students what is involved in a constructive code review process. Hundhausen, Agrawal, and Agarwal discuss this kind of in-person review, dubbed *pedagogical code review*, in early courses [13]. In pedagogical code review, a small group led by a moderator use a set of predefined coding practice guidelines to guide a group review of student programs.

### 6.4 Review Feedback (Meta-Reviewing)

Any use of peer-review must choose whether to include grading or feedback on the contents of reviews themselves. We use the term *meta-review* to refer to any feedback on a review (because feedback can be considered a review of a review). Feedback can take many forms: the author who received a review could report on whether the review was constructive or led to changes, course staff could formally grade reviews and return comments to the reviewer, or third parties could comment on the relative merits across a set of reviews. Which model makes sense depends on factors including the learning objectives for IFPR, features of peer-review software, and course logistics (such as staff size relative to student population). Many of the issues here apply to peer-review in general, rather than only to IFPR.

According to Ramachandran and Gehringer [41] reviews consist of (1) summative, (2) problem detection, and (3) advisory content. Meta-reviews can report on each of these three types of contents, each of which is valuable in its own way. While summative contents can reflect a reviewer's understanding, problem-detection content directly helps a student identify opportunities for improvement, and advisory content points out ways in which students might improve. Meta-reviews can include information on which parts of a review were constructive, and which led to actual changes. Meta-reviews written by authors of submissions can also include rebuttals to aspects of a review: in IFPR, such rebuttals can arise when students are debating the requirements of an exercise through the review process (a healthy outcome relative to the goals of IFPR). In the case of rebuttals and follow-ups, this blurs the line between collaboration and review, as students may end up coming to a shared understanding that may not have occurred with a single review step.

#### 6.4.1 Types of Meta-Reviewing

Around Ramachandran and Gehringer's framework, there are several ways to structure the information in meta-reviews, and provide useful feedback to reviewers.

- **Direct feedback from course staff**: Feedback from instructors or TAs can repair incorrect advice and reinforce good behavior (case studies 1, 11, and 13 call out the importance of correcting faulty reviews explicitly). Class size is clearly a factor here. If someone wants to use peer-review to help scale human feedback in large courses, then giving expert feedback on reviewing might not be feasible.
- **Feedback based on assessment of submissions**: In situations where an expert evaluation of the *assignment* is available (whether through auto-grading or by human TA) and reviews are quantitative, it should be possible to automate a meta-review that tells a reviewer something about the quality of work they reviewed. For instance, if a student indicates in a Likert scale that they "strongly agree" that a solution is correct, but the grade for the assignment they reviewed is low, an automated meta-review can indicate that this review likely mis-evaluated the work under review.
- **Reporting correspondence among reviews**: Reviewers could be told about the correspondence between their evaluation of a submission and those of other students. For example, the SWoRD tool for peer review of writing tells student reviewers, on each criterion they reviewed, how they did relative to the average of other students' scores [7]. An example of feedback that they show says "Your ratings were too nice for this set of papers. Your average rating was 6.50 and the group average was 5.23." This hints to the reviewer that he may have missed something in his review. This does run into issues of calibration and opinion: just because a student disagrees with the average, it doesn't mean they are wrong! The outlying reviewer may have understood something the other reviewers didn't, in which case comparing their review to an expert's, or to a trusted automated process, may be more useful feedback.
- **Having students review submissions of known quality**: In CaptainTeach programming assignments, half the time students are asked to review a known-good or known-bad solution (implemented by the course staff) [22]. Students use a Likert scale in each review to indicate whether they think the submission under review is correct: if a student gives a strong score to a known-bad solution, or a weak score to a known-good solution, she gets immediate feedback telling her of the discrepancy.

Existing research has explored ways to provide or assess meta reviews. Nelson and Schunn describe a rubric for evaluating peer feedback in writing assignments, which includes criteria like the concreteness and actionability of the review, and whether it was generally positive or negative [35]. Swan, Shen, and Hiltz study assessment strategies for comments in online discussion forums used to discuss class content [51]. Though the discussions are not necessarily critiques of student work—they are simply prompts for questions and comments—they do have similar requirements to reviews in relevance, accuracy, and focus.

The Expertiza peer review process contains an explicit review-of-review phase for collaborative work [40], and a related Expertiza tool attempts to give some more qualitative feedback automatically by a naturaly language analysis of student work [41].

#### 6.4.2 Using Meta-Reviews

While one generally may prefer to eliminate low-quality contents in reviews, in a pedagogical context receiving some low-quality review contents can be beneficial. While in traditional educational settings students may trust all the feedback they receive from the instructor, in IFPR students have to learn to assess the value of the reviews they receive. They will have to learn to triage review comments into those they will act upon and those they will ignore. Moreover, having a diversity of reviews, maybe even contradictory ones, can be a starting point for valuable discussions in class. Having to wade through reviews can implicitly train reviewers that they, in turn, should not submit "brain dumps" of everything they think of, but instead provide valuable and concise reviews: that the important metric is actionability, not volume.

Instructors may seek to use meta-reviews to monitor the IFPR process. Given the quicker turn-around times inherent to IFPR, such monitoring benefits from tool support and structural elements of meta-reviews. For example, asking students to rate the reviews they receive on a simple Likert scale makes it easy for an instructor to focus on potentially problematic reviews without undue burden on the students. In some IFPR configurations, software tools that include automatic grading could report partial information on whether student performance improves following the review phase. Such information would be most useful for identifying cases in which poor work did not improve, prompting the instructor to check on whether the author had received useful and actionable advice through reviews.

Meta-reviewing incurs a cost. Whether or not meta-reviews are worth that cost depends on the learning goals: if teaching how to review is important, meta-reviews are essential; however if the learning goals focus on artifact production or performance, and if the reviewers are experienced, meta-reviews may be less essential. An alternative to providing meta-reviews for each review is to provide a few example reviews and their meta-reviews. To not tempt students to simply reuse the best example review comments, these exemplar reviews can come from an assignment that is different from the current assignment.

A live demonstration of how to do a code review is a form of scaffolding on the process-level, but does not drive content as specifically as rubrics. Regardless of how reviewing is introduced and scaffolded, it is important to allot time to deal with misconceptions on how to create a review as part of the course design.

## 7. Parameters and Issues

Several issues and parameters cross-cut the stages of the IFPR process discussed in section 6. Questions about preventing plagiarism, integrating IFPR with course-level grading, deciding where to use anonymity, involving experts, making IFPR relevant for non-majors, engaging students in the process, and identifying software needs all guide one's particular configuration of IFPR. We discuss each of these issues in turn.

### 7.1 IFPR and Plagiarism

IFPR, like many course and assignment structures, requires careful mechanism design to ensure that students aren't incentivized towards detrimental behavior that lets them get a good grade at the cost of their (or others') education.

One of the most immediate problems with IFPR is that, by definition, students are shown one another's work while in the middle of an assignment. Since the final submission happens after students have been exposed to other students' work, the IFPR educator must determine how to account for this exposure when assigning a grade to the final submission.

At the extreme, a student could submit an empty initial submission, copy what he sees during the reviewing phase, and submit the copied solution as his own final solution. In less extreme cases, a student may copy all or part of another solution into her own after submitting an initial first try that she becomes convinced is incorrect. There are a number of course- and grading-design decisions that can affect the degree to which copying is a problem:

- **Variation in Assignments**: One major factor in whether copying is even a problem is how similar students' submissions are expected to be. In many programming courses, students implement to the exact same algorithmic specification; other than coding-style issues, one implementation is just as good as another. This is in contrast to other domains where peer review is often used, like creative or critical writing, in which students often write on different topics or choose different positions to represent on the same topic.

  One approach is to provide different variants of a programming problem to different students: Zeller [60] gives each student a variation on a theme to avoid students reviewing another who is working on exactly the same problem. Indeed, it is often possible to generate large numbers of different problems automatically from a specification, as Gulwani et al. have done for algebra problems and more [2, 47].

  A drawback of variation in assignments is that it weakens one of the benefits of IFPR – having students review the same problem they are already thinking about! Especially for beginning students, where program comprehension skills are still being learned, one goal of IFPR is to lessen the cognitive load of the comprehension task by having the student review code for a problem they already understand. If they have to internalize an additional problem description along with new code, this puts significantly more overhead into the reviewing process.

  Depending on the learning goals, it may be good for the reviewer to learn to incorporate ideas from different solutions into their own, since it requires a more abstract understanding of the techniques. For novices, it may be enough of a challenge to recognize a good solution apply it to their own.

- **Weighted Submission Grading**: There is often value in having students copy parts of other submissions that they see in order to improve their own work. It happens all the time in professional software development, and the act of recognizing a good solution demonstrates understanding that is far beyond blind plagiarism. Students should take things from the examples they see and demonstrate that they learned from them; however, a student has no guarantee that what he is seeing is correct, so blindly copying can hurt!

  However, wholesale copying (where a student submits an empty file then copies the best of what they see) should be discouraged. In order to mitigate this, Politz et al. [22] grade IFPR assignments by assigning heavier weights to initial submissions than to post-review submissions: an initial program submission counts for 75% of the grade. Students can still improve the

25%-weighted part of their score based on review feedback and copying others' solutions, but they can also hurt their score if they make incorrect changes. Different weightings put different emphasis on the importance of review. Having the post-review score count for more might be acceptable in some classroom settings, and ultimately comes down to a choice about student maturity, class culture, and other course-specific factors.

- **Alternative or Supplemental Grading**: Another solution to the grading problem is to supplement assignment grading with other techniques that cannot be copied. For example, in an in-person code review of a student's solution, an instructor can quickly ascertain whether the student has simply copied something or actually understands the code they have submitted. This can be done by, for example, asking the student to change their program to match a new specification, or asking her to understand a proposed change to her submitted code.

### 7.2 Interaction with Course-Level Grading

Instructors must determine the extent to which IFPR activities impact course grades, and the mechanisms through which they do so. Section 1 noted Liu and Carless' distinction between *peer feedback* and *peer assessment* [34], where the latter's goal is grading. Most of the working group discussion focused on peer-feedback (which fit the course contexts of the participants), though we gave some attention to peer-assessment (more often proposed to address grading at scale in large courses).

#### 7.2.1 Peer Assessment and IFPR

Kulkarni et al. have shown that, with careful rubric and mechanism design, peer assessment can produce similar results to TA grading in MOOCs [33], and Reily et al. report on the accuracy of a combination of peer reviews at assessing programming assignments [42]. Peer assessment changes the motivation structure of IFPR. For example, a student who is afraid of affecting their peers' grades with negative feedback may be more hesitant to give that feedback. In contexts where students are still learning to review and give feedback, inaccurate reviews are expected and an important part of the learning process: in this case, reviews probably should not be used for grading purposes. Using peer review for grading should be adopted with care, and practitioners should carefully consider its effects on the other design decisions discussed in this report.

#### 7.2.2 Should Reviews Be Graded?

Although section 6.4 discussed various forms of feedback on reviews, it did not discuss whether reviews should be assigned scores that affect students' course grades. Grading schemes can range from checkbox-style points for submitting reviews (without grading content), to more detailed assessments. Of our case studies, six (3, 5, 6, 8, 9, and 11) explicitly tie reviews to the course or assignment grade in some form. No case study discussed grading meta-reviews.

In general, the group members were reluctant to ascribe grades to reviews (as opposed to giving meta-reviews, which the group strongly endorsed). The group shared concerns that having reviews influence course grades (beyond required participation) would misdirect student motivation for reviewing [31].

Nonetheless, the group did discuss options for having reviews figure into grades. We discussed basing assignment grades on reviews rather than on the work submitted (on the grounds that reviews reflect students' understanding of the assignment): this would allocate staff grading time to meta-reviews rather than to code, which could be more valuable (as some, though not all, aspects of code can be assessed automatically). We also discussed basing students' grades on the improvements that their reviews inspired in the work of others, but felt the nuances (submissions with little room for improvement, students who chose not to act on reviews) made this infeasible.

#### 7.2.3 Interaction with Relative and Curve Grading

The group noted that IFPR (like other collaborative course structures) interacts poorly with grading strategies that evaluate students relative to one another. Such strategies conflict with students' motivations to help one another improve their work. The working group identified three distinct ways in which the conflict could manifest:

- **Demotivating**: There is disincentive to do good review, because it can push others past oneself in achievement, adversely affecting one's own grade.
- **Destructive**: Instead of just being apathetic about review, students could even sabotage one another with bad feedback, hoping to reduce others' scores to actively improve their own.
- **Unmotivating**: Since the curve puts a limit on how much one can achieve, there is a disincentive to *respond* to feedback or reflect (e.g. especially for high-achieving students, there's little reason to take feedback seriously).

No one in the group used relative grading in their own courses, so we lacked first-hand experience in mitigating these problems in that context.

### 7.3 Anonymity

Several issues relating to anonymity and privacy come to the fore with peer review. Developing a culture of positive and constructive critique, where students can both give and take feedback appropriately, can take time and require a degree of practice. While the broader aim may be to develop a positive, supportive and professional approach to peer review, there may be a need to provide some initial shielding from scrutiny for students who are new to the institution or the practice.

#### 7.3.1 Types of Anonymity

Each of the IFPR roles—authors and reviewers—can be anonymous to either or both of other students and faculty. Reasonable arguments can be made for each configuration, and different configurations have been used in practice.

In PeerWise [14], a system to which students submit proposed study questions on course material, student submissions are ranked for quality and correctness and the content of the contributed questions is public. In that case, not revealing contributor identities to peers is important: this creates a degree of safety for the novice student contributor and helps identify the public ranking with the work and not with the student. However the contributor identities are visible to the instructors as student contributions may be summatively graded. A variant on this (similar to some conference reviewing models) could also see reviews made visible to all reviewers, which may help in establishing and reinforcing norms and standards and mitigate the risks of unduly harsh or abusive reviews.

In case study 10, anonymity is staged: at first students review one another anonymously, then groups are formed and groups review one another, revealing identities and adding a social element. This lets students do a first round of reviewing to get comfortable with the process of feedback, and after has the benefits of encouraging professional collaboration.

In total in our case studies, only two (4 and 11) explicitly stated that reviews were anonymous. Several (studies 5, 6, 8, 12, and 15) had reviews that were in-person, and therefore cannot be anonymous. The others left it unstated, and in different course contexts the assignments could be administered either way.

### 7.3.2 Upsides of Anonymity

At the introductory level, students suffer from both confidence and maturity problems, making anonymous review an attractive option (at least initially). Students will not face personal embarrassment if others see work that they are not comfortable showing publicly, and anonymity gives reviewers the freedom to be more candid.

In addition, students who know the student under review might make assumptions based on the student rather than the submission. This could even cause students who consider themselves weaker to not question incorrect work that comes from a supposedly smart student. Anonymity helps level the playing field in the face of such preconceptions.

In general the group considers anonymity to be a less jarring initial option for IFPR that is more likely to protect students who are new to the peer-review process. However, sharing identity during review has significant benefits in the right contexts, and anonymity isn't without its own problems.

### 7.3.3 Downsides of Anonymity

Anonymity can unwittingly enable a culture of 'flaming' and online harassment to develop among some students. Therefore if reviews are to be done anonymously, policies and techniques for reporting abuse and inappropriate behavior should be in place – for example, an anonymous "Flag abuse" button in a Web interface that allows students to bring offensive or inappropriate content to the attention of the staff. In general, the issues with anonymity relate to the unsatisfactory aspects of peer assessment schemes in group work that ask group members to evaluate relative contributions [10].

The benefits of non-anonymity center mainly around creating collaborative cultures and helping students learn professional behavior. Non-anonymity creates opportunities for students to acknowledge each others' contributions. As a broader educational goal, ethics and professionalism are meant to be covered as part of our curricula [18]; an open model for peer review gives a clear opportunity for enforcing appropriate behaviour. Industrial code reviews are not done anonymously [44], so students gain relevant skills from learning to give and receive non-anonymous feedback.

### 7.3.4 Anonymity and Cultural Considerations

Anonymity may have cultural connections at several levels: for students who come from a consensus-based national culture [27], where preserving harmony is a strong value and overt criticism can be considered offensive or cause a loss of face, a greater level of anonymity may be required at first for students to feel more at ease in speaking their minds.

A converse cultural aspect may be in operation with non-anonymous contributions: if students know the other person, they may be less or more critical a priori, based upon judgements of the peer's relative status or perceived expertise, rather than reviewing the material in its own right.

At some institutions such as Brown University (where three of the group members have studied IFPR), full anonymity is already hard to establish, because a robust undergraduate TA program means that students often act as TAs for one another independent of peer review, and know about one another's performance. At Brown, the institutional and student culture makes it commonplace to know who is reading your code.

The group debated whether giving students choice in anonymization was a good idea, and concluded that it was not desirable because it encouraged hiding attitudes, and may make people justify being more objectionably critical because they were allowed to be anonymous. A further negative was that it would not bring the shy students out (which is sometimes the educator's goal). It was concluded that the preferable approach was to make anonymity or identifiability a matter of policy, and educate students about the importance of professionalism in either case.

Overall the group doesn't recommend that IFPR practitioners adopt anonymity by default, but rather that they take course and culture into account. Overly stressing anonymity could unwittingly give the impression in students that peer review is dangerous. We do recommend that in cases where work is identifiable, the underlying goals and expectations related to professionalism and community-building be consciously introduced to students from the outset. This could be talked about as explicitly and strongly as the discussions about plagiarism, with potential penalties for abuse of the system through lack of respect for one another.

## 7.4 The Role of Experts

Peer review can sometimes usefully be complemented with expert review. There are several arguments for and against combining the two. Potential benefits of expert review include: moderating conflicts between reviewers and reviewees; facilitating contribution and collaboration; overcoming cultural adjustment problems; and providing exemplars of good work and good reviewing.

- **Experts as Moderators and Facilitators**: Experts can act as moderators to make sure that issues and conflicts that arise, whether in a live situation or asynchronously, can be dealt with by an authority figure. As moderators, experts do not take on the role of reviewers, which keeps students in charge of the feedback itself. So moderation is a form of process rather than content expertise (akin to pedagogical code reviews [13], which are led by an expert moderator). Another form of process expertise arises when experts are present not for moderation but as facilitators. Students who are not sure of themselves may not contribute much; experts can assist in getting students to contribute, and push the idea of review as a learning process, in addition to acting as figures of objective authority on grading.

  Expert reviewers can help address cultural issues where students devalue the opinions of their peers and overvalue the opinions of instructors. They can also offset deficiencies in the knowledge and insight of students who are themselves adjusting to the process of giving informed critiques, and ensure that some knowledgeable, high quality feedback is received.

- **Perception and Quality of Expert Review**: Expert review (especially instructor or TA provided) may have unwarranted special status in students' minds: feedback from experts may be interpreted as "more relevant for my grade," and hence more likely to be acted upon. This last concern is called out explicitly by case study 9, in which students produce a peer-reviewed research article, and in the past often discounted peer feedback in favor of instructor-provided comments.

  Cho and MacArthur hide the provenance of expert- vs. peer-provided reviews in order to study whether expert review is in fact more effective at improving students' grades [8]. They compare three approaches to giving feedback on written assignments in a psychology course: feedback from a single peer, feedback from a single topic expert, and feedback from multiple peers [8]. The results indicate that feedback from multiple peers results in better quality revisions than feedback from an expert, with feedback from a single peer being the worst. The hypothesised reason for this was that peers gave feedback that was phrased in terms that students could more easily comprehend. It certainly would be interesting to see whether the same results apply to programming assignments.

- **Expert-provided Exemplars and Models**: In IFPR the goal is to avoid attaching the idea of constructive review to an expert being present. However, an expert can provide models for review that students can follow. So one strategy would have experts give examples, or be present for some (early) sessions and not others. These exemplars and supports represent a form of scaffolding reviewing.

The working group members discussed an "Editor's picks" option, where instructors and TAs highlight good examples of reviews for others to learn from. TAs might monitor reviews and post a handful of high-quality ones for the entire class. As a tweak to avoid simply externally rewarding good examples (which comes with the downsides of extrinsic motivation [31]), the reviews picked don't necessarily all need to be objectively good. Instead, the editor's picks could highlight interesting reviews, and what is interesting (or could be better!) about them. Students then learn specifically why something is considered good or bad in a review, and can apply that knowledge to their own reviewing. The process of breaking down the review shifts the emphasis from quality of reviewer to important aspects of the review itself.

## 7.5 Does IFPR Make Sense for Non-Majors?

When instructing computer science majors, reviews of code can be motivated by their resemblance to code reviews and similar activities in the software profession, even though this may not be the actual driver behind their inclusion in a course. When using in-flow peer review with non-majors, such motivations might work less well due to a disconnect between the students and the IT profession. Regardless of the future profession, we view the reviewing skill as an important one and note that reviewing comes in naturally in a number of fields, from academic writing to zoology, but also in the students' everyday lives in reviews of movies and restaurants on websites, etc.

We frame in-flow peer reviewing as a technique for improving learning based on timely feedback throughout a process. This can be different from a means of quality assurance of a final artifact, or a technique for learning to produce and consume reviews, although these concepts arise naturally in a setting where in-flow peer review is used. Knowing how to produce and consume reviews is a useful skill in its own right, regardless of the profession or the context in which it is used. On this note, using reviews in classes with a diverse student population will have the additional benefit of producing more diverse reviews. In many cases, for such as GUI mockups, this is highly desirable. Therefore, whether or not a student will face a review professionally is not necessary for IFPR to be useful.

Tests are useful for non-majors to review as they are often less tangled with complicated "non-functional aspects" of the problem such as memory management, performance, or code elegance. Case study 1 features an assignment from a course designed explicitly for non-majors, with review of tests. We also envision that non-majors reviewing code might reduce "code fear" by making code less magical and establishing that code can be wrong. This is also related to the "community of practice" of programming in that students are seeking to join the broader community of programmers by taking a CS course in the first place. Giving non-majors a sense of what code—and programmers—are doing might teach respect for code and programming, while not necessarily trying to make students into professional programmers. Many programmers are managed by non-programmer experts in other fields who could benefit from an understanding of practices surrounding code. There are also cases where non-programmers must be able to read or relate to code to make sure that code follows complicated legal practices or financial algorithms.

Finally, as an important aside, reviewing might help with retention especially by removing the stigma of computer science being an anti-social, "inhuman" subject. Humanities people generally read more than they write – as reviewing is more like reading, it might feel more comfortable for them than programming. This could be a component of a CS course in the style of fine arts, which Barker et al. found to create a better community culture and encouraged retention of female students [30].

## 7.6 Bringing Students Along

One challenge of peer review in general lies in getting students to value the contributions of their peers. Peer evaluations potentially contradict a model students have of the instructor as the sole authority. In addition, many students aren't initially comfortable acting as reviewers (and giving criticism to one another!), so a desire for harmony may hinder their ability to produce effective reviews at first. These two dimensions—taking the reviews seriously and taking act of reviewing seriously—differ in force and mitigations.

When students are in the role of reviewers, instructors may need to consider how to balance anonymity in reviewing with the need to engage students in the professional practice of reviewing (section 7.3). When addressing students coming from cultures that don't emphasize review and criticism, explicit reminders that review is part of the working programmer's life in western culture can be helpful.

When students are in the role of receiving reviews, instructors should require students to demonstrate engagement with the reviews. Asking students to indicate how they used reviews in revising their work is one option. Students may need explicit instructions on how to read reviews; this might also help them cope with criticisms. For some students, peer review may be their first time getting negative feedback; instructors should make response to review a positive activity. For students whose work was strong and did not yield actionable reviews, an instructor could ask "what have you learned from looking at others' solutions?" Hopefully, such activities on a few early assignments would help students develop self-reflection skills.

Specific advice one could give students includes:

- When you receive criticism, go back to other reviews – are you making the same mistake over and over?
- Have a cooling-off period if you get negative reviews – you may view them more positively with a little passage of time.
- Do you see a contradiction in your reviews? Maybe one of the reviewers is wrong, or maybe your work isn't clear enough.
- How will you avoid making these mistakes again in the future? Look at the review, break out the actionable items, prioritize.
- Are you taking the review personally? Remember that the review is about the work (and improving it!) not about you – the ability to recognize your weaknesses and improve them is incredibly valuable. This is an attitude encouraged by *egoless programming* [57].

If students remain skeptical of the value of peer-review, a mid-course survey on the value of reviewing might help convey the experiences of others. This also fosters a culture of peer-review as a collective effort rather than one of criticism, judgment, or assessment.

## 7.7 Software and Analytics for IFPR

Good software tools are key to making IFPR manageable and informative for both students and course staff. For students, tools that integrate reviewing with the IDE used for programming assignments, for example, mitigates some of the context switching that the process otherwise requires. For staff, good tools not only manage the logistics of the process, but can also be instrumental in producing meta-reviews and in monitoring the effectiveness of IFPR.

Software systems could (1) give basic feedback on review quality by comparing reviews between students [7] or through use of machine learning methods [41], (2) aggregate student responses for discussion [26], (3) flag students who consistently write certain kinds of reviews (weak, strong, superficial, etc.), or (4) maximize variety of review tasks by using static analysis tools to appraise source code similarity. Some of these features are more sophisticated than those included in current peer-review platforms.

Features for organizing submissions or reviews in various ways could help instructors run effective IFPR processes. Software could allow instructors to associate arbitrary tags with each submission or review, such as "sloppy" or "discuss in class," "ignore," "insufficient test coverage," or "misunderstood requirement B." Instructors could then search or group submissions and reviews by tags, or use tags as categories when producing analytical visualizations. Tags could be visible only to instructors, or instructors could make tags visible to authors or reviewers. Another way to organize submissions or reviews would be to cluster them automatically by various criteria, similar to Expertiza's automatic meta-reviewing categories [41]. Example criteria include the length of the reviews or their tone ("positive," "neutral," or "negative"), which may be detectable automatically using natural language processing techniques. Based on this automatic and manual organization, instructors could identify recurring misconceptions, or they could select representative exemplars of reviews to be presented in class. Finally, the information in reviews (e.g., feedback in each rubric, or scores on Likert scales) could be used to organize submissions, and information in meta-reviews could be used to organize reviews.

Instructors as well as students could benefit from analytic visualizations or dashboards. Visualizations presenting activity over time could help to understand the timeliness of reviews or a student's progress over time in terms of the quality of their submissions or their reviews. Visualizations of reviewer-author relationships, which could include various historic aspects, such as the quality of their past submissions or the usefulness of their past reviews, could help with reviewer assignment. Visualizations superimposing reviews on top of submissions or meta-reviews on top of reviews could provide a compact picture of a certain artifact to authors, reviewers, or instructors.

## 8. Related Code-Review Practices

Much of our discussion of IFPR has revolved around code review, as programs are one of the most common artifacts in computer science courses. Given the rich history of industrial code-review practice, it is worth asking what IFPR can learn from this practice. We look at two forms of industrial code review: pair programming and peer code review.

### 8.1 Pair Programming and IFPR

Pair programming (henceforth PP) is a software-development technique in which two programmers work together on one computer. PP involves significant (and continuous) in-flow peer feedback, though coding together is a rather different activity than writing and responding to reviews. One surface-level difference lies in the number of reviewers: IFPR students may receive multiple reviews, whereas comments in PP come from a dedicated programming partner. Other interesting differences arise along three dimensions: responsibility, skills developed, and dynamics. To better align the practices, we contrast PP with an IFPR model called "mutual review" (discussed in section 6.3) in which a pair of students are tasked with reviewing one another's submissions.

- **Responsibility**: The key difference between mutual review and PP is that in PP both students are responsible for the quality of a *single* artifact, whereas in IFPR each student is responsible for her own artifact. This can naturally lead to a significant difference in motivation and responsibility.

  The differences in shared responsibility can also be seen when pairing students of different strengths. In PP, if one student is stronger, that student may drive the production of the artifact, or even take over the work, without the weaker student having the opportunity to participate fully; thus, the weaker student may not benefit from the experience. In IFPR, a strong student's assignment (and hence grade) is not affected by the weak student. The strong student is in a better position to help the weaker student; although though the strong student may receive no beneficial comments, no harm will be done. With IFPR, any problems due to mismatched pairings can be alleviated by assigning multiple reviewers. In practice (in several authors' experience), in PP students often work alone even when expected to work collaboratively.

- **Skills**: The skills required and developed differ between IFPR and in PP. PP deals primarily with the collaborative creation of an artifact. IFPR focuses also on the creation of an artifact, though less collaboratively as students do not work face-to-face. IFPR also focuses on the high-level skill of performing reviews, which requires both program comprehension and judgement. These activities require students to take a higher-level perspective on the task at hand, including the difference between understanding one's own code and understanding the code of others. IFPR also forces students to trade off work, namely, time spent on their own assignment versus time spent on reviewing. This trade-off is not present in PP. To continue the analogy with academic paper writing and reviewing: PP is more like working as coauthors, whereas IFPR is more like the relationship between a journal paper reviewer and an author (especially if there are multiple IFPR stages).

- **Dynamics**: The difference between solo and shared responsibility for the artifact yields different dynamics between PP and IFPR. Due to the continuous communication required in PP, students are often immediately alerted to problems in program comprehension. In contrast, IFPR is separated by both space and time. As a result, programs can acquire significant accidental complexity. Students realize this, and thus learn about the difficulties of producing and reading code, both by trying to make sense out of the submissions of others and by seeing the feedback their own submissions receive.

  Finally, with IFPR, students are compelled to review other students' code: they cannot ignore problems and let their partner do the work. On the other hand, in IFPR it may be easier to ignore the advice given than in PP, because of the difference in ownership – it is harder to argue that someone not make suggested changes to a shared program than to ignore the feedback on one's own program.

### 8.2 Industrial Peer-Review Practice

Code review is an essential component of industrial software-development practice. Industrial peer review is inherently in-flow, as reviews are conducted on a regular basis during development – it would be odd indeed to only perform code review after a product had shipped to customers! The industrial product-development life cycle is longer than that in many courses, but best practices in industrial peer review are still useful context for this report.

#### 8.2.1 Motivations for Industrial Code Review

Industrial code reviews differ in motivation from pedagogic code reviews. The goal is often to reduce the defect rate before releasing software or committing to a design. This is the primary measure of an effective review process in Fagan's seminal work [15], in followup work to it [19], and also in some modern surveys tied to large case studies [11]. The goals of in-flow peer review in pedagogic settings are much broader than just finding bugs in peers' code, though finding problems is certainly one worthy cause for review.

However, in one modern study that studies attitudes about code review in both developers and managers at Microsoft, researchers found that defect finding, while important, was only one of several top motivations developers saw for review [4]. 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 indicated specific cases where they learned about a new API that they could use in their own code, or providing links to the code author with documentation of other alternatives, suggesting that these activities do indeed occur.

Activites like knowledge transfer and dissemination of ideas are certainly goals of in-flow peer review, and in-flow strategies should consider ways to foster them. Bachelli and Bird [4] note that these metrics are harder to measure than defect rate, but observe them coming up spontaneously in interviews and in observations of reviewers.

#### 8.2.2 Staged Code Inspections

Fagan's seminal work on code inspections in an industrial setting [15] 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.

This result mirrors our intuitions about the value of staging assignments for review at points that are useful for catching and fixing misconceptions about the assignment. Our primary goal is not simply to improve the programming output of students, however. We care about what they learn from seeing other examples, teaching them to effectively review others' code, and more. Still, it is useful to consider that the in-flow experience is similar to effective industry practices, and note that professional developers benefit from the staging process.

#### 8.2.3 Meetings vs. Asynchronous Code Review

Fagan's original results are for *formal code inspections* [15], 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 [19]. Votta concludes that much of the benefit of code review can be had without the overhead of scheduling in-person meetings.

There is further research on this debate, but the only clear conclusion is that significant benefits of review remain even without in-person review. In a pedagogic setting, in-person reviews may serve other goals, like training students to review, encouraging productive feedback, and supporting the social aspects of review. However, industry research suggests that the overhead of scheduling and holding meetings isn't a prerequisite of effective reviews in professional settings.

#### 8.2.4 What and How to Review

A large industrial case study on code review [11], which also documents a survey of code review in industry (including a longer discussion of formal vs. lightweight review), identifies guidelines for effective reviewing practices. By measuring defect rates found against the number of lines of code under review and the length of the review session, their study recommends "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."

## 9. Additional Related Work

### 9.1 Intrinsic vs. Extrinsic Motivation

Ways in which particular assignment designs tradeoff between intrinsic and extrinsic motivation have been a theme in this report. The group frequently considered arguments from Kohn's "Punished by Rewards" [31], which argues that praise and rewards (such as grades) are ineffective and even demotivating. Kohn claims that rewards effectively punish students who aren't rewarded, discourages risk-taking in students, devalues the reasoning behind a low or high grade since the grade is emphasized, and can create a bad relationship between teacher and student. IFPR is designed to increase intrinsic motivation for reviewing relative to traditional post-submission peer review. The extent to which this shift occurs likely interacts with how the grading system treats reviewing.

### 9.2 Metacognitive Reflection

One goal of in-flow review is to encourage reflection while in the middle of an assignment. Meta-cognitive reflection has been studied as an important part of the learning process. Indeed, it has been indicated that one difference between experts in program comprehension and novices is the focus on metacognition [16]. Others in different contexts have found effective reviewing and prompting strategies for encouraging reflection that IFPR can learn from.

Palinscar and Brown study *reciprocal teaching*, in which a teacher alternates with a student in a dialog that prompts for reflective activities, like generating summaries or clarifying confusing elements [50]. They used reciprocal teaching with seventh graders struggling with reading comprehension, with an emphasis on letting students take over as the session progresses. The entire point of the exercise is to encourage reflective activities in students, and similar prompts—for summarization or clarification—may lead students in IFPR contexts to give reviews that cause more reflection, or more directly reflect themselves.

Davis and Linn [1] use explicit self-review prompts at different stages of assignments given to eighth graders. For example, in one assignment, students had to perform a repeated task (designing clothing and environments to help cold-blooded aliens survive). They compared responses to direct prompts submitted along with a design, like "Our design will work well because...", to prompts designed to encourage reflection after the fact, like "Our design could be better if we...", and plan-ahead prompts designed to cause reflection during the assignment, like "In thinking about doing our design, we need to think about..." Their sample size was small, and they did not find a significant difference in design quality between the direct and reflective prompts. They did find that students gave better explanations when given the reflective prompts, but the difference could easily be attributed to the small sample size.

Frederiksen and White have done a series of studies on *reflective assessment* and *reflective collaboration* in middle school science classes [39, 59]. In an online environment, students work on mock experiments using a scientific-method like flow for a project: they start with an initial inquiry, form hypotheses, analyze mock data, and draw conclusions. In between steps, they are asked questions that urge them to reflect on their work: why they think a hypothesis is true, if they are being meticulous in analyzing their results, and more. In addition, the environment contains simple autonomous agents, called advisors, that give automated feedback and suggestions to students. They are also given the opportunity to assess the work of other students in a few ways: they can simply rate the contents, or they can make specific suggestions, like telling a student that they should pay attention to a particular advisor. Finally, the course is complemented with role-playing activities where students take on the role of advisors, and give specific feedback – the advisors have specific flavors of feedback; an example in the paper is that a student acting as the "'Skeptic' might [be asked to] say 'I disagree' or 'Prove it'" to another student's submission.

### 9.3 Learning From Examples

In IFPR, students have the opportunity to take what they learn from examples of others' work and apply it to their own. There is existing research in how students take what they learn from existing examples (whether from peers or experts) and use it to learn principles they apply to new problems.

There is a large body of work that studies learning from *worked examples* [3], in which students learn solely or primarily from example solutions to problems with accompanying descriptions of the problem solving process used. Worked examples are traditionally created by experts, and in IFPR the examples students see come from their peers. In addition, in IFPR students see the examples *after* they attempt the problem themselves. Finally, worked examples generally have explanations at a finer granularity than we would consider presenting peer review at. Nevertheless, some of the recommendations of the worked examples literature may be relevant in the IFPR setting: in a broad survey of worked examples research, Atkinson et al. recommend that students "experience a variety of examples per problem type," and that examples be presented in "close proximity to matched problems" [3]. Both of these recommendations are consonant with strategies we have proposed for IFPR assignments.

Chi et al. [53] and Renkl [43] studied the effects of asking students to provide talk-aloud self-explanations along with worked examples, and found that students who were better at producing self-explanations of the examples performed better on post tests. This correlation was deemed to be independent of many other factors, including domain knowledge of the students coming into the task. A peer review is a kind of explanation, and may cause students to go through the same kind of reflective process when studying other students' examples.

Kulkarni et al. [32] discuss changes in creative output between subjects who saw varying numbers of examples, and diversity in examples, prior to creating their own artwork. Subjects seeing more diverse examples created artwork with more unique features than subjects seeing a less diverse set or fewer examples. In an IFPR setting, students who see examples from others, especially when already primed to think about the same problem, may similarly have more options to draw on in their solution, rather than only using whatever techniques they would have tried in their initial submission.

In PeerWise [14], students created and reviewed one another's multiple-choice questions, which has elements both of learning by example and of review. They find that students who engaged with the system more—by contributing and explore more example questions than others (and more than they were required to by the course)—performed better than those who did not. However, it's not clear that the exposure to more examples caused the difference in performance.

### 9.4 Peer Instruction

Peer Instruction (PI), which is a specific form of student-centered pedagogy [37], has been shown to be a promising way to improve student performance [23] and engagement [46] both in introductory courses [23] and upper-division courses [5]. Peer instruction, as defined by Crouch et al. [23], focuses on engaging students in activities that require them to apply the core concepts under study and to explain these concepts to their peers. Concretely, a class taught using PI principles can consist of short presentations, each of which focuses on a particular core concept, which is then tested by presenting students a conceptual question, which the students first solve individually and then discuss in groups.

Related to our undertaking, the most interesting component of PI is the peer discussions that are undertaken after presentation of each concept. This differs from (in-flow) peer review in that there isn't a submission in question that students are trying to improve. The exercises are a vehicle for discussion, and the discussion is the end goal, not producing a quality submission. Neither of these goals is necessarily more helpful than the other, but depending on expected outcomes, one can be more effective. In programming and writing disciplines, for example, one explicit goal is to train students to produce quality programs and written work. In mathematics or physics, it may be more important that students understand concepts and know how and where to apply them, rather than producing any particular artifact.

### 9.5 Comprehending Program Structure

Program comprehension is at the same time a prerequisite and a learning goal of in-flow peer review of programming assignments. Students need some ability to read code in order to provide a meaningful review to one another, but at the same time, IFPR can lessen the cognitive burden of comprehension by having students review problems that are conceptually close (or identical) to something the reader has just encountered. The degree to which it tends toward one direction or another is a function of the experience level of the students and the goals of the particular course.

In addition, for in-flow assignments centered around programming, one goal of peer-review is to help students reflect on their own code structure. There is a rich literature on program comprehension (including contrasting experts and novices), but much of that focuses on understanding the *behavior* of a new program [17, 45, 49]. In the IFPR context, students already know the problem and (roughly) what the program is supposed to do. Reading others' code therefore has different goals: notably, to understand the structure that someone else brought to the problem and to contrast that with one's own. This is a less burdensome task than asking if the program *matches* an existing specification.

Studies on program comprehension comparing experts and novices have found that experts engage in more metacognitive behavior [16], so the metacognitive context of review may put students in the right frame of mind to understand programs in the first place. Other work on program comprehension suggests that the process has a lot to do with understanding the high-level plan of a program [49]. Since in the in-flow context students have at least constructed *a* plan of their own for the same or a similar problem, they may at least be able to determine if the solution they are viewing matches their plan, or is doing something different.

### 9.6 Increasing Socialization in Programming-Oriented Courses

IFPR has potential to foster a collaborative and social atmosphere in programming assignments. We discussed some of the motivations for a more social CS course when discussing IFPR for non-majors (section 7.5). There are other approaches to meeting these goals that IFPR can learn from.

Garvin-Doxas and Barker emphasize the importance of the classroom climate, emphasizing that courses that reward "hero" programmers and individual accomplishment give rise to a defensive atmosphere that can be counterproductive to learning for students with less prior ability [20]. In later work, Barker and Garvin-Doxas describe the outcome of running an IT course more like a fine arts course than a traditional engineering course [29]. This included projects that were more meaningful, public critique of results, and routing collaboration. This approach created a classroom culture where learning is a social and community process, rather than isolated, and the result was a greater retention of female students than the traditional engineering teaching approach.

The technique from Barker et al. most relevant for in-flow peer review, though not completely the same, is the approach to knowledge sharing during lab work. Students actively solicit help from any student, for example, by yelling questions out, resulting in a fluid exchange of ideas and techniques. Even though the projects considered in these labs were run in an open, collaborative setting, cheating was avoiding by using individualized assignments.

Another working group discussed design decisions in computer-mediated collaborative (CMC) educational settings [58]. That report emphasizes goals that CMC can help reach, including encouraging peer review, having teamwork experiences, developing self-confidence, and improving communication skills. A course using IFPR that performs review through an online tool is certainly an instance of a CMC setting, and addresses many of the same goals.

### 9.7 Existing Uses of In-flow Peer Review

Others have used strategies for peer review that fall under the umbrella of in-flow peer review, even though they did not go by that name.

In the implementation of a multi-stage compiler, students in Søndergaard's course review one another's work between stages [52]. 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 [40] (discussed in section 6.4.1) is used for large, multi-stage collaborative projects. This includes assessment of the reviews themselves as an explicit motivator for giving good feedback. It is notable that in Expertiza, students often review other students' components of a larger whole, which can be a task that the reviewer didn't complete him or herself. In several of our case studies (1, 3, 4, 13, and 14), students review an instance of the *same* work that they just did themselves.

CaptainTeach [22] supports in-flow peer review for programming assignments. The Web-based tool supports test-first, data-structure-first, and one-function-at-a-time stagings of programming problems. It uses asynchronous reviewing, where students see 2-3 reviews from the most recent students to submit, combined with random known-good and known-bad solutions provided by the staff. It supports a fixed set of open-ended review prompts combined with Likert scales for each of tests, implementation, and data structures. Students are also allowed to give (optional) feedback on reviews they received. Politz et al. report that students engaged with the process, submitting stages early enough to get reviews (more than 24 hours before the deadline), and receiving review feedback promptly (a few hours) [22]. In another analysis of data on reviews of test suites in CaptainTeach, Politz et al. report that students were more likely to add missing tests for a feature after review if they reviewed or were reviewed by a student who had tested for that feature [21].

Informa's "Solve and Evaluate" approach integrates a simple form of peer review into a software-based class room response system [26]. During a lecture the instructor poses a problem, and each student solves it by creating a solution in Informa. Informa is not limited to multiple-choice problems; it also allows a variety of problem types, including free text (e.g., code snippets), or drawings (e.g., diagrams of the structure or state of a program). Students submit their solution as soon as they are done, and they immediately are assigned a solution of a peer for evaluation. They evaluate a solution simply by scoring it as correct or incorrect. In Informa, a key reason for including an evaluation phase is to keep the faster students engaged while the slower students are still solving the problem. While the results of peer review are not shown to the authors of the submissions, they are used by the instructor to estimate the level of understanding of the class, and to focus the class discussion that follows the evaluation phase. A lecture using Informa often consists of multiple stages, and often the problems in subsequent stages build on each other (e.g., the first problem asks students to draw a control-flow graph of a program with conditionals, and the second problem asks for a control-flow graph including loops). In such a scenario, a lecture with Informa is an instance of in-flow peer review.

### 9.8 Actionable Peer Review

Some other uses of peer review on large projects are related to in-flow peer review because they allow students to improve their work in response to review. These uses don't necessarily stage assignments into reviewable pieces, instead performing review on entire intermediate artifacts. For example, Clark has students exercise the functionality of one another's projects, and lets groups improve their work based on the feedback their classmates give them [9]. Similarly, Wang, et al. [56], Zeller [60], Papadopoulos et al. [36], and the Aropä system [25] use assignment structures that allow students to update revisions of entire submissions that were reviewed by peers. Other studies have students write test cases (or manually test) one another's work as part of a review [42, 48]. These tests are most often on entire systems, rather than on pieces of a project that build up along with reviews. Students do, however, have the chance to improve their projects in response to their peers' feedback.

## 10. Conclusion

IFPR is a highly-configurable mechanism for making peer review more actionable. It leverages the fact that many problems, both programming and otherwise, can be split into several steps that occur at key moments for triggering reflection, and uses those moments as vehicles for peer feedback. It encourages reflective and critical thinking in both reviewers and reviewees, and prepares students for professional activities in judging others' work and incorporating feedback into their own.

IFPR has a lot in common with existing peer review approaches, and in collaborative and participatory pedagogic styles in general. All of those benefits, from enhancing a sense of community to improving communication skills, are also goals of IFPR. The main new idea is to engage students in the collaborative process by better integrating into the flow of assignments.

This report outlines a large space for designing in-flow peer review assignments. We encourage practitioners to consider many of these factors, but not to be intimidated by them, or to be concerned that there are too many challenges to tackle in adopting IFPR. The key task is to pick good moments for reflection in the middle of assignments, and use those moments to get the most out of peer feedback (which we already know has many benefits).

## Bibliography

[1] Elizabeth A. Davis and Maria C. Linn. Scaffolding students' knowledge integration: prompts for reflection in KIE. *International Journal of Science Education* 22(8), 2000.

[2] C. Alvin, S. Gulwani, R. Majumdar, S. Mukhopadhyay. Synthesis of geometry proof problems. In *Proc. AAAI Conference on Artificial Intelligence*, 2014.

[3] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research* 70(2), pp. 181–214, 2000.

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

[5] Cynthia Baily Lee, Saturnino Garcia, and Leo Porter. Can Peer Instruction Be Effective in Upper-division Computer Science Courses? *Trans. Comput. Educ.* 13(3), pp. 1–22, 2013.

[6] B. S. Bloom and D. R. Krathwohl. Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I: Cognitive Domain. Longmans, 1956.

[7] Kwangsu Cho and Christian D. Sun. Scaffolded writing and rewriting in the discipline: A web-based reciprocal peer review system. *Computers and Education* 48, pp. 409–426, 2005.

[8] Kwangsu Cho and Charles MacArthur. Student revision with peer and expert reviewing. *Learning and Instruction* 20(4), pp. 328–338, 2010.

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

[10] Tony Clear. THINKING ISSUES: Managing Mid-project Progress Reviews: A Model for Formative Group Assessment in Capstone Projects. *ACM Inroads* 1(1), pp. 14–15, 2010.

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

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

[13] 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.

[14] Paul Denny, John Hamer, Andrew Luxton-Reilly, and Helen Purchase. PeerWise: Students Sharing Their Multiple Choice Questions. In *Proc. Proceedings of the Fourth International Workshop on Computing Education Research*, 2008.

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

[16] Anneli Eteläpelto. Metacognition and the Expertise of Computer Program Comprehension. *Scandinavian Journal of Educational Research* 37, pp. 243–254, 1993.

[17] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Mental Representatinos of Programs by Novices and Experts. In *Proc. INTERACT*, 1993.

[18] Ursula Fuller, Joyce Currie Little, Bob Keim, Charles Riedesel, Diana Fitch, and Su White. Perspectives on Developing and Assessing Professional Values in Computing. *SIGCSE Bull.* 41(4), pp. 174–194, 2010.

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

[20] Kathy Garvin-Doxas and Lecia J. Barker. Communication in Computer Science Classrooms: Understanding Defensive Climates As a Means of Creating Supportive Behaviors. *J. Educ. Resour. Comput.* 4(1), 2004.

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

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

[23] Catherine H. Crouch and Eric Mazur. Peer instruction: Ten years of experience and results. *American Journal of Physics* 69(9), pp. 970–977, 2001.

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

[25] John Hamer, Catherine Kell, and Fiona Spence. Peer Assessment using Aropä. In *Proc. Australasian Computing Education Conference*, 2007.

[26] Matthias Hauswirth and Andrea Adamoli. Teaching Java Programming with the Informa Clicker System. *Science of Computer Programming*, 2011.

[27] Geert Hofstedet and Gert Jan Hofstede. Cultures and Organizations: Software of the Mind. McGraw-Hill, 2005.

[28] Simon Hooper and Michael J Hannafin. The Effects of Group Composition on Achievement, Interaction, and Learning Efficiency During Computer-Based Cooperative Instruction. *Journal of Educational Computing Research* 4, pp. 413–424, 1988.

[29] Lecia J. Barker, Kathy Garvin-Doxas, and Eric Roberts. What Can Computer Science Learn from a Fine Arts Approach to Teaching? In *Proc. Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005.

[30] Lecia J. Barker, Kathy Garvin-Doxas, and Eric Roberts. What can computer science learn from a fine arts approach to teaching? In *Proc. SIGCSE Technical Symposium on Computer Science Education*, 2005.

[31] Alfie Kohn. Punished By Rewards. Houghton Mifflin Company, 1999.

[32] Chinmay Kulkarni, Steven P. Dow, and Scott R. Klemmer. Early and Repeated Exposure to Examples Improves Creative Work. In *Proc. Cognitive Science*, 2012.

[33] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia , Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. Peer and Self Assessment in Massive Online Classes. *ACM Transactions on Computer-Human Interaction*, 2013.

[34] Ngar-Fun Liu and David Carless. Peer feedback: the learning element of peer assessment. *Teaching in Higher Education* 11, pp. 279–290, 2006.

[35] M. M. Nelson and C. D. Schunn. The nature of feedback: How different types of peer feedback affect writing performance. *Instructional Science* 27(4), pp. 375–401, 2009.

[36] Pantelis M. Papadopoulos, Thomas D. Lagkas, and Stavros N. Demetriadis. How to Improve the Peer Review Method: Free-selection vs Assigned-pair Protocol Evaluated in a Computer Networking Course. *Comput. Educ.* 59(2), pp. 182–195, 2012.

[37] Eric Mazur. *Peer Instruction: A User's Manual*. 1996.

[38] Amanda Miller and Judy Kay. A Mentor Program in CS1. In *Proc. ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2002.

[39] John R. Frederiksen and Barbara Y. White. Cognitive Facilitation: A Method for Promoting Reflective Collaboration. In *Proc. Computer Support for Collaborative Learning*, 1997.

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

[41] Lakshmi Ramachandran and Edward F. Gehringer. Automated Assessment of Review Quality Using Latent Semantic Analysis. In *Proc. IEEE International Conference on Advanced Learning Technologies*, 2011.

[42] 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. Proceedings of the ACM International Conference on Supporting Group Work*, 2009.

[43] Alexander Renkl. Learning from Worked-Out Examples: A Study on Individual Differences. *Cognitive Science* 21, pp. 1–19, 1997.

[44] Dieter Rombach, Marcus Ciolkowski, Ross Jeffery, Oliver Laitenberger, Frank McGarry, and Forrest Shull. Impact of Research on Practice in the Field of Inspections, Reviews and Walkthroughs: Learning from Successful Industrial Uses. *SIGSOFT Softw. Eng. Notes* 33(6), pp. 26–35, 2008.

[45] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An Introduction to Program Comprehension for Computer Science Educators. In *Proc. Proceedings of the 2010 ITiCSE Working Group Reports*, 2010.

[46] Beth Simon, Sarah Esper, Leo Porter, and Quintin Cutts. Student Experience in a Student-centered Peer Instruction Classroom. In *Proc. Proceedings of ACM Conference on International Computing Education Research*, 2013.

[47] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *Proc. AAAI Conference on Artificial Intelligence*, 2012.

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

[49] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions of Software Engineering* 10(5), pp. 595–609, 1984.

[50] Annemarie Sullivan Palinscar and Ann L. Brown. Reciprocal Teaching of Comprehension-Fostering and Comprehension-Monitoring Activities. *Cognition and Instruction*, pp. 117–175, 1984.

[51] Karen Swan, Jia Shen, and Starr Roxanne Hiltz. Assessment and Collaboration in Online Learning. *Journal of Asynchronous Learning*, 2006.

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

[53] Michelene T. H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Riemann, and Rober Glaser. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science* 13, pp. 145–182, 1989.

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

[55] Lev Vygotsky. Interaction between learning and development. *Mind and Society*, pp. 79–91, 1978.

[56] Yanqing Wang, Hang Li, Yanan Sun, Jiang Yu, and Jie Yu. Learning outcomes of programming language courses based on peer code review model. In *Proc. International Conference on Computer Science & Education*, 2011.

[57] Gerald M. Weinberg. The Psychology of Computer Programming. Van Nostrand Reinhold, 1971.

[58] Ursula Wolz, Jacob Palme, Penny Anderson, Zhi Chen, James Dunne, Göran Karlsson, Atika Laribi, Sirkku Männikkö, Robert Spielvogel, and Henry Walker. Computer-mediated Communication in Collaborative Educational Settings (Report of the ITiCSE '97 Working Group on CMC in Collaborative Educational Settings). In *Proc. The Supplemental Proceedings of the Conference on Integrating Technology into Computer Science Education: Working Group Reports and Supplemental Proceedings*, 1997.

[59] Barbara Y. White, John R. Frederiksen, T. Frederiksen, E. Eslinger, S. Loper, and A. Collins. Inquiry Island: Affordances of a Multi-Agent Environment for Scientific Inquiry and Reflective Learning. In *Proc. International Conference of the Learning Sciences (ICLS)*, 2002.

[60] Andreas Zeller. Making Students Read and Review Code. In *Proc. ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2000.

## 11. Appendix – Links to Case Study Details

For readers interested in the details of the case studies summarized in figure 2 and figure 3, the table in figure 4 maps case-study numbers (from the leftmost column of the summary tables) to filenames within a public git repository. The repository URL is in the figure caption. Each detailed summary contains notes on how the case-study author envisions configuring IFPR, including thoughts on problem staging, rubric design, and grading considerations.

| **Study #** | **Link** |
|---|---|
| 1 | krishnamurthi/asgn-1.md |
| 2 | clarke/asgn-1.md |
| 3 | politz/asgn-1.md |
| 4 | hauswirth/asgn-1.md |
| 5 | wrigstad/asgn-1.md |
| 6 | wrigstad/asgn-2.md |
| 7 | tirronen/asgn-2.md |
| 8 | clarke/asgn-2.md |
| 9 | clear/asgn-1.md |
| 10 | fisler/asgn-1.md |
| 11 | fisler/asgn-2.md |
| 12 | hauswirth/asgn-2.md |
| 13 | krishnamurthi/asgn-2.md |
| 14 | politz/asgn-2.md |
| 15 | tirronen/asgn-1.md |

Figure 4: Links to the full descriptions of case studies on the Web (see https://github.com/brownplt/iticse-in-flow-2014/tree/master/in-flow-assignments).