



Barriers for Students During Code Change Comprehension

Justin Middleton
Department of Computer Science
North Carolina State University
USA
jamiddl2@ncsu.edu

John-Paul Ore
Department of Computer Science
North Carolina State University
USA
jwore@ncsu.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
USA
ktstolee@ncsu.edu

ABSTRACT

Modern code review (MCR) is a key practice for many software engineering organizations, so undergraduate software engineering courses often teach some form of it to prepare students. However, research on MCR describes how many its professional implementations can fail, to say nothing on how these barriers manifest under students' particular contexts. To uncover barriers students face when evaluating code changes during review, we combine interviews and surveys with an observational study. In a junior-level software engineering course, we first interviewed 29 undergraduate students about their experiences in code review. Next, we performed an observational study that presented 44 students from the same course with eight code change comprehension activities. These activities provided students with pull requests of potential refactorings in a familiar code base, collecting feedback on accuracy and challenges. This was followed by a reflection survey.

Building on these methods, we combine (1) a qualitative analysis of the interview transcripts, activity comments, and reflection survey with (2) a quantitative assessment of their performance in identifying behavioral changes in order to outline the barriers that students face during code change comprehension. Our results reveal that students struggle with a number of facets around a program: the context for review, the review tools, the code itself, and the implications of the code changes. These findings – along with our result that student developers tend to overestimate behavioral similarity during code comparison – have implications for future support to help student developers have smoother code review experiences. We motivate a need for several interventions, including sentiment analysis on pull request comments to flag toxicity, scaffolding for code comprehension while reviewing large changes, and behavioral diffing to contrast the evolution of syntax and semantics.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution*; **Software development process management**; • **General and reference** → **Empirical studies**.

ACM Reference Format:

Justin Middleton, John-Paul Ore, and Kathryn T. Stolee. 2024. Barriers for Students During Code Change Comprehension. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3597503.3639227>

1 INTRODUCTION

Code review is a useful and popular software engineering practice wherein team members manually inspect each other's new code to verify that it meets expectations before integrating it into the official product [4]. Not only does code review improve quality by catching bugs early [41], but it also promotes organizational cohesion by spreading project knowledge throughout the team [11]. To attain these benefits, code review requires comprehending the differences between the current program version and the new proposal. However, isolating the behavioral differences underlying syntactic differences is an error-prone process for professionals and students alike [33], and obstacles in the review process can trickle into the project's health altogether. For example, prior work finds that professional programmers experience confusion during code review when the code is unfamiliar or they are provided no rationale for the change, and these deficiencies delay development [12].

Therefore, because of review's centrality to professional software process and the variety of ways it can be poorly applied, many software engineering education courses incorporate code review to prepare students for industry practices [22, 24]. Students and professionals respond to different pressures in different contexts, and the prior literature is split on where and to what extent professionals and students diverge. Some studies suggest they behave differently when it comes to code review and code comprehension tasks. For example, a body of prior work finds that novices investigate software with different or less effective patterns than experts [9, 25, 28], but the impact of experience is not uniformly underlined throughout contexts [39, 40]. Other studies suggest that student and professional developers face similar challenges when comparing similar algorithms, and both struggle to identify behavioral differences [33]. These ambiguities make it more difficult to identify effective interventions across contexts.

Regardless of whether there are differences between the populations, support code comprehension during code review is needed across the board [11], especially when it comes to *refactoring review* [3]. Research efforts at improving the expressiveness of programmatically generated reports of difference, or diffs, support this notion, such as in their augmentation with runtime information [14]. By studying student developers and the barriers they

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

ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04

<https://doi.org/10.1145/3597503.3639227>

encounter, we can compare their barriers to professional developers and, like the curb cut effect [23], unveil needs with solutions that could bring benefit to all developers.

Our investigation focuses on junior-level undergraduate students enrolled in a software engineering course. It involves interviews, tasks, and a post-activity survey aimed to understand what is hard about understanding code changes. The interview focused on identifying code review barriers that may not be immediately discoverable through a prescribed activity. We designed code change comprehension tasks as proposed refactorings through GitHub pull requests, where participants judged if each refactoring was done correctly (i.e., preserves behavior). This task design focuses on the comprehension aspect of the code change and has a clear measure of accuracy. Students then reflected on the experience in a post-activity survey. We found four high-level categories of barriers faced during code change comprehension—context, tool, code, and comparative comprehension—comprising 13 barriers in all. Thus, this work makes the following contributions:

- A data-driven characterization of code change comprehension as a challenging task for students, and
- Identification of 13 barriers students face when comprehending code changes during pull request review.

Despite differing contexts, we find some commonalities in the barriers that students face when comprehending code changes with those faced by professionals in an industrial context [12]. Using the barriers as the outline of an underlying process, we also explore the cognitive activities of comparison and opportunities for improving student developers' experience of it with additional tools.

2 RELATED WORK

To motivate our work, we track developments in the practice of peer code review and its integration into classrooms.

2.1 Code Review in Professional Settings

The term *modern code review* (MCR), was popularized by Bacchelli and Bird. Through observations, interviews, and surveys at Microsoft, they highlight code reviews' perceived benefits: to uncover defects, improve code, propose alternative solutions, and disseminate knowledge [4]. Davila and colleagues [11] corroborate these benefits with a systematic review of MCR, aggregating technical factors (e.g., code patch features) and nontechnical factors (e.g., author and reviewer traits) that influence review outcomes.

MCR's implementation can change with the context. Sadowski and colleagues explore MCR at Google, mixing investigative methods to corroborate the generality of some traits of MCR (e.g. MCR is tool-driven and lightweight) while finding areas of variability (e.g. formatting consistency is an explicit priority in their context) [44]. Outside of corporate industry, Rigby and colleagues focus on open-source software (OSS) projects, interviewing developers and compiling review policies and artifacts [42]. Convergent results include two reviewers being sufficient; divergent results include the self-selection process of reviewers; Rigby and Bird report additional comparisons in their syntheses of industry and OSS alike [41].

Germane to our interest in behavioral preservation, AlOmar and colleagues performed a case study at Xerox to discover refactoring's place in MCR [2]. In their assessment, refactoring often copes with issues in design and style but often occurs alongside functional

changes, too. They find that refactoring documentation is often insufficient and propose a framework of three I's (*Intent, Instruction, and Impact*) to mitigate challenges in industrial refactoring.

Our methodology borrows from prior work in industry through use of interviews and surveys, but we add code change comprehension tasks instead of direct observation, allowing us to better explore the accuracy with which code changes are understood.

2.2 Code Reviews Barriers

Kononenko and colleagues identify two categories of review barriers: technical and social. The former includes code unfamiliarity, size, and complexity; the latter, time management and task switching [27]. The barrier of large changes is also present in Baum and Schneider's call for novel review tools, which recommends interventions such as reducing changeset sizes and supporting or obsoleting complete code comprehension [6]. Pascarella and colleague's interviews and assessment of code review comments highlight seven categories of review-relevant information needs, including alternative solutions, rationale, and whether the review is decomposable into smaller changes [36], suggesting that the lack of meeting those information needs could create a barrier.

Building on their aforementioned industrial case studies, AlOmar and colleagues also contrast code review with refactoring review in open-source projects. They detail new categories of obstacles for refactoring review, such as confirming behavioral preservation or otherwise untangling refactorings from intentional behavioral changes [3]. Likewise, they quantitatively corroborate that refactoring reviews generate more comments and code churn than reviews otherwise. These results suggest that refactoring review may present unique challenges.

Ebert and colleagues combine OSS developer surveys with the mining of code reviews to categorize sources of confusion like inadequate rationale or familiarity [12]. This methodology in the OSS sphere reflects our interests within a student population, and discovering overlap and distinction can resolve the debate on whether and where novice and professional populations diverge.

2.3 Code Review in Education

Given code review's value to professional practice, it draws attention in education research. For example, Trytten experimented with a class review activity and reported successes like exposure to new algorithmic approaches, as well as challenges like hostile discussions [50]. Song and colleague's PCR is a web-based tool to assist student reviewers with code rubrics, and they find that student participants value the ability to see alternative solutions to familiar problems [47]. On the other hand, Chong and colleagues experiment with having students develop their own code review checklists, which is an activity that could supplement instructors with more insight into how students scaffold the review process than measuring defect detection alone [10]. They report that a majority of the checklist questions that students generate are useful for reviews, and those questions that are not often struggle with clarity or independence from specific testing or analysis tools.

From a refactoring-specific perspective, Keuning and colleague's experimented with a web-based tutoring system and found varying

success over six refactoring exercises [26]. However, to our knowledge, code review of refactorings, as we do in this study, has not been studied in a student population.

Indriasari and colleagues frame code review as a field-specific implementation of peer review, and summarize the literature that explores classroom implementations of it [24]. On the one hand, they corroborate peer review to be an effective supplement in classrooms for encouraging coding standards, constructive communication, and time management. On the other hand, the authors report common, general barriers to review efficacy, such as the lack of knowledge, low engagement, low review quality, and ineffective administration and review processes. Our research shares a common concern with the barriers to review; we supplement this literature, however, by probing the comprehension process in particular in addition to probing students' experiences outside the classroom.

2.4 Comparative Comprehension

Code review and refactoring both involve the contrast between versions of code: the original version and the updated version. Hence, they are acts of comparative code change comprehension. Tao and colleagues' survey highlights that developers and test engineers practice change understanding several times a day, whether for reviews or feature additions and bug fixes [48]. Likewise, one of AlOmar's refactoring review challenges is the distinction of refactoring changes from non-refactoring changes [3]. These studies highlight the challenge of tracking code behavior while comprehending code changes, and so we want to bring that into focus with our observational study.

The frame of review as comparison also takes special meaning in educational situations. One benefit of code review is to see alternative methods for solving a problem [36, 47, 50], illustrating that witnessing and comparing alternative solutions may have educational benefits. This notion is corroborated by studies like Patitsas and colleagues [37], which find that studying alternative algorithms in parallel can improve students' knowledge and flexibility. Studies that find explicit benefit in this practice suggest that there is something unique occurring in comparative comprehension in contrast to other kinds of software comprehension.

In prior work by two of the authors of this research, Middleton and Stolee [33] measured the accuracy with which student and professional developers identify code clones. They also interviewed participants about the practical situations in which code comparison arises; refactoring and code review emerged as primary contexts. In this work, we also look at the accuracy of comparison but in context, through a refactoring lens, and with students only. Further, we focus on the barriers they face, as these can lead to future interventions.

3 METHODOLOGY

The primary goal of this study is to understand the challenges that student developers (hereafter just "*students*") endure when comparing versions of a program during code review. In this frame, we ask two research questions. The first is this:

RQ₁ What barriers do student developers face when comprehending code changes?

To answer this question from multiple perspectives, we consolidate three sources of qualitative data. First, in interviews, we ask

students to explicitly discuss their negative experiences in code review, or explicitly discuss their ideal workflow. Second, we elicit *in-situ* reflections during a code change comprehension activity where participants examine specific proposed refactorings. Third, we distribute a post-activity reflection survey so that students can combine their specific experiences in the activity with their general experience otherwise.¹

While the initial interviews can elicit topics from the complete process of peer review, the change comprehension activity and reflection survey do not in themselves replicate the end-to-end process of code review, which is collaborative, team-based, and long-term. Rather, we focus on an essential step within code review—the examination of self-contained code changes—delivered via pull request in a familiar code base by students.

For our second research question, our focus is to assess the potential impact of barriers. After all, barriers can have negative downstream effects on developer performance: prior work suggests that professionals experience delays, decreased code quality, and negative emotion when barriers in review emerge [12]. Given the limited time scale of our activity, we can focus on one of the most immediate of consequences: correctness in judgment. That is, prior work also shows that developers struggle to identify behavioral differences in similar algorithms [33]. Thus, we operationalize our construct of *comparative comprehension* in this research as *the accuracy with which a developer detects behavioral differences resulting from a code change*. Therefore, to measure the potential impact of such barriers unveiled by RQ₁, we pose another question:

RQ₂ How accurately do student developers recognize behavioral impact in code review tasks?

For answering RQ₂, we use the code change comprehension task to measure participants' accuracy in identifying refactorings (behavior preserving) and non-refactorings (behavior non-preserving).

3.1 Study Context

We designed this study for a junior-level undergraduate Software Engineering course at North Carolina State University. When we administered the study in Fall 2021, the course had 130 students. This course required all students to have experience in the fundamentals of computer science and Java. Furthermore, this course incorporated code review using GitHub with Git branches, pull requests, and automated testing during a team-based course project that spanned the last eight weeks of the term.

Specifically, the course project centered on a web portal prototype called iTrust v2 [32]. The project repository contains 287 HTML files, 149 Java files, and many other auxiliary files. Each team of three to four students wrote, integrated, and tested a significant new feature to the project. Each team submitted and merged 32-35 pull requests, demonstrating sufficient experience with the pull request interface and process. Given that this course project preceded our study's interviews and class activity, all students had familiarity with team-driven code review and the code.

¹This work is IRB approved under NCSU 24542. Per this approval, anonymous participant quotes can be published but raw data and transcripts cannot be released. However, study artifacts are available to facilitate replication: <https://zenodo.org/records/10145856>.

- (1) **Background questions (repeated in post-activity survey):**
 - (a) Are you familiar with the concepts of code refactoring? → Please define in your own words.
 - (b) And code review, especially with a team? → Please define in your own words.
 - (c) What have been your experiences in receiving code reviews?
 - (d) What experiences have you had outside of class?
- (2) **Review Techniques questions:**
 - (a) When doing code review on new or changed behavior, what tools and techniques did you have to determine behavior?
 - (b) Are those techniques good enough, or do you wish you had a better way?
 - (c) Do you typically get to see both versions of the code at the same time?
- (3) **Quality & Refactoring Questions**
 - (a) Have you made suggestions during code review to improve code quality without changing the overall behavior?
 - (b) How do you define code quality in these situations?
 - (c) What techniques did you have to determine quality?
 - (d) What techniques did you have to determine if behavior has been maintained?
 - (e) Do those techniques typically work as intended and in a timely manner, or could they be better?
- (4) **Speculative Questions**
 - (a) If you had access to any information or tool you'd like, what would be your ideal way of examining, comparing, and reviewing code?
 - (b) What are your biggest sources of frustration?
 - (c) What are other times you compare code, beyond review?

Figure 1: Semi-Structured Interview Questions for RQ₁

Toward the end of the semester, we announced an upcoming class activity and directed students to an online consent form. All students would participate in the activity during class, but their consent determined if we could analyze their responses. We also invited them to participate in 15-minute interviews before the class activity in exchange for extra course credit.

3.2 Interviews

We conducted the one-on-one interviews over Zoom, which automated the first draft of transcription. Each interview lasted up to 20 minutes. Our interview questions are in Figure 1, but the execution was semi-structured: we reordered major topics and created follow-up questions in response to the flow of conversation. Our major topics involved their understandings of refactoring and review (#1-2), the processes they had experienced for reviewing code or assessing code quality (#3), and their biggest frustrations and most desired changes for those processes (#4). Interview participants received one extra percentage point of class credit.

3.3 Code Change Comprehension Activity

To create the tasks, we forked the iTrust v2 [32] repository and manually explored it for refactorings opportunities from Fowler's text [17] and Murphy-Hill and colleagues' study of professional refactoring [34]. Given that many reviews in professional practice do not perfectly separate small changes into separate commits [35], we included more than one refactoring in many examples. We split

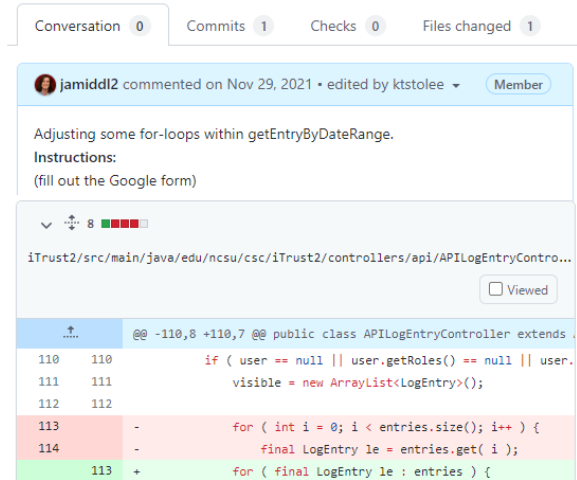


Figure 2: An example of the interface which students used to review refactorings. The top half shows a basic pull request description. The bottom half shows the changed code.

the tasks into two groups, one with real refactorings, and one with behavioral changes, corroborating both with the project test suite. Thus, we created refactorings in 11 non-overlapping areas in iTrust v2, using Eclipse's automated refactoring tools when possible.

Table 1 lists all tasks, noting which were refactorings (white background rows) and behavioral changes (grey background rows). The first three, A, B, and C, were INTRODUCTORY examples that were walked through as part of the introduction. For example, task C introduces a behavioral change by replacing `Stream.findAny` with `Stream.findFirst`. The change involves one file (# FILES); +9 lines are added and -3 lines are removed. Three tasks were refactorings (tasks 3, 5, and 6) and three were not (tasks 2, 4, and 7); these six PRIMARY tasks answer RQ₂. We intended the remaining two tasks to be refactorings, but after conducting the activity, we discovered ways that these changes impacted behavior that was not covered by existing tests. Rather than throwing these tasks out altogether, we therefore labeled these exercises as OPEN-ENDED and we removed the correctness scores (tasks 1 and 8).

We presented the code change comprehension tasks to students using the pull request review interface on GitHub. The overall task instructions framed each code change as a potential refactoring, and we designed the pull request description to briefly describe the area of change without revealing the ground truth. Figure 2 provides an example of what the participant would see. The change proposal is captured in Pull Requests with a "Files Changed" page. Obsolete code is highlighted in red, and updated code is highlighted in green. This corresponds to Task 2 in Table 1. For each of the eight tasks, participants responded to four comprehension questions shown in Figure 3; all questions were optional.

After the code comprehension activity, students completed a reflection survey, designed to elicit reflections from the activity they had just completed. The questions are shown in Figure 4.

3.4 Study Execution

After announcing the upcoming class activity and distributing consent material, we interviewed 35 student developers over four days,

Table 1: Details of the Code Change Comprehension Activity.

Shaded rows are non-refactorings (code edits that change behavior). “DK” means “I do not know.” “%Correct” excludes “I do not know” responses. An arrow (→) in a refactoring implies that “Program Structure 1 was converted into Program Structure 2.”

TASK	CODE CHANGES	IS REFACTORING?	#FILES	#LINES	#	Y	N	DK	%CORRECT	
INTRODUCTORY										
A	Rename Variable (x2)	Yes	1	+5	-5					
B	Exception → Precheck	Yes	1	+2	-11					
C	Stream.findAny → Stream.findFirst	No: new code ensures deterministic behavior.	1	+9	-3					
PRIMARY										
2	for loop → for each loop (x2)	No, new code maintains references to an obsolete variable.	2	+3	-5	42	35	<u>7</u>	0	16.7%
3	Loop → Pipeline	Yes	1	+6	-7	43	<u>34</u>	2	7	94.4%
4	Consolidate Conditional Extract Variable (x4)	No: behavior differs when comparing two null values.	1	+10	-40	44	38	<u>5</u>	1	11.6%
5	Consolidate Conditional Extract + Move Function	Yes	2	+35	-22	44	<u>31</u>	10	3	75.6%
6	Replace Magic Literal (x4) Stream.collect.size → Stream.count (x4)	Yes	1	+21	-13	38	<u>27</u>	4	7	87.1%
7	Extract Function (x2) Slide Statement	No: the extracted methods contain early return statements.	1	+110	-102	31	22	<u>2</u>	7	8.3%
OPEN-ENDED										
1	Extract Variable Rename Variable (x2)	No issues when run alone, but issues in a surrounding framework that depends on parameter names.	1	+12	-10	43	41	2	0	
8	Exception → Precheck Extract Function Extract Variable (x4) Inline Variable (x2) Loop → Pipeline Slide Statement	For some cases of invalid input, the order in which errors are raised may change, but this is compensated for in other files.	1	+30	-37	29	21	3	5	

- (1) **Is this code change a refactoring (i.e. it does not change the external behavior)?** {"Yes", "No", "I do not know"}
- (2) **What impact does this code change have?** If it is a refactoring, does the refactoring improve the legibility, maintainability, or something else? If it is not a refactoring, what behavior does this pull request change? If you do not know, what feature of the code are you unsure about?
{free text}
- (3) **What tools or strategies did you use to investigate the differences in the code?** Explain in a comment what strategies you used in comparison. Did you use an IDE? Did you use the unified or split view in GitHub? Did you run the test suite? All of the above? None of the above? *{free text}*
- (4) **Was there anything difficult about comparing this code?**
{free text}

Figure 3: Questions for each code change comprehension task; (1) and (2) address RQ₂; (3) and (4) address RQ₁. Italicized texts are the possible answers.

spending 10 to 20 minutes per participant. While the interviews were conducted outside of the class time, the activities and reflections were conducted during class.

We began the 75-minute class with a lecture introducing technical debt, code smells, and refactoring (“*semantics-preserving transformations on source code*”). After 25 minutes, we moved from lecture

- (1) What was difficult about performing the code review in this study?
- (2) What was easy about performing the code review in this study?
- (3) What would have helped you perform the code review in this study more effectively?
- (4) How many years of programming experience do you have?
- (5) Do you have experience in professional software environments? With code review? With refactoring?
- (6) What is your gender identity? *[male/female/non-binary/prefer not to disclose]*

Figure 4: Reflection survey questions answered at the end of the study; used to answer RQ₁.

to the walkthrough of the three introductory tasks, A, B, and C. Because we performed these tasks as a class, tasks A, B, and C are not included in the results.

After 10 minutes talking through the exercises, we directed the class to a Google form which linked to the other statically ordered tasks 1–8 on our iTrust v2 fork on our university enterprise GitHub. Participants had 30 minutes to work through as many tasks as their pace allowed. All participants saw the same task ordering so that we could collectively reflect on the first three tasks as an end-of-class activity, although they could skip among questions.

At the end of class, we directed students to the reflection survey. We allowed students to return to the form later if they wanted to

Table 2: Demographics of participants. Interview participants are a subset of task participants. For gender identification, no students selected “non-binary” or opted out.

	Interviews	Tasks + Reflection
Sample Size	29	44
Gender		
Male	19	31
Female	10	13
Programming Experience (Years)		
Range	2 to 9	2 to 9
Mean	4.1	4.0
Median	4	4
Experiences in Professional Environments		
Any	17 (58.6%)	23 (52.3%)
Review	12 (41.4%)	18 (40.9%)
Refactoring	11 (37.9%)	17 (38.6%)

complete more tasks on their own time. To account for the potential effect of additional time, we marked the submission times as during or after class.

3.5 Participants

Of the 35 interview participants, only 29 submitted their demographic information in the Reflection Survey at the end of data collection. We summarize their details in Table 2 in the *Interviews* column, which is a proper subset of the activity participants (Section 3.3). The range of programming experience in years was between 2 and 9, with an average of 4.14. Over half of respondents (17 of 29) reported experience in professional environments such as internships and co-ops. We note that future work should consider established instruments for measuring expertise to better understand the role it plays in comparative code comprehension [15].

After the class activity, we distributed an online form to collect four kinds of data: participants’ programming experience in years, whether they had professional experience with code review, their biggest challenges with the class activity, and their personal demographic information. Of the 130 students in the class, 75 students consented to data analysis, 58 students submitted answers for the class activity, and a separate but overlapping set of 61 students submitted demographic information. We had 44 usable class activity records from participants who submitted all three. Their general demographics are included in Table 2 and, when quoted, we refer to them as P01 through P44 with random but consistent identities.

Because we allowed students to submit their forms after class, we also analyzed whether these data would be comparable to those in class. Of the 44 students, 9 submitted after class. The mean number of correct tasks for in-class submissions was 2.40, and the mean after-class was 2.44. A Welch’s t-test of these data does not provide sufficient evidence for us to conclude that the scores were significantly affected by this decision ($p=0.88$).

3.6 Data

For RQ₁, the data come from three sources: the interviews before the class activity (29 participants answering question in Figure 1), free-text responses after every task (132 descriptions by 44 participants);

and the reflective survey after the class period (44 summaries by 44 participants).

For RQ₂, from the 44 participants we received 314 total responses to the code change comprehension tasks, an average of 7.14 from each. Thirty responses were “I don’t know” for their respective code change. We exclude responses to tasks 1, 2, and 3 from one participant because their open-ended responses referred to the three introductory tasks A, B, and C instead.

3.7 Analysis

For RQ₁, the first author generated a set of categories an initial open-card sort [29, 43] (i.e. no pre-existing categories) by doing an initial pass over the full set of interviews, activity answers, and class reflections. Another author then took these codes and applied them to only the class reflections, noting places of disagreement. We then convened, discussed new categories that could resolve disagreements, and discussed the axial categories which could further organize the codes. With the new codes, we applied them to the rest of the data.

To measure interrater agreement after independent coding, we use Cohen’s Kappa [31]. When one or both raters assigned multiple codes to a single quote, we duplicated the quote into multiple observations, pairing shared codes with each other and unshared codes likewise, to prevent one area of disagreement from overruling co-occurring areas of agreement. After the initial, independent coding of class reflections to establish the shared codebook, our kappa value was 0.59. After deliberation and applying the codebook to the full dataset, our kappa value was 0.67, representing a moderate to substantial level of agreement. We deliberated over disagreements until we resolved all of them.

For RQ₂, we grade participant correctness—whether or not the change preserves behavior, as described in Section 3.3—based on our ground truths in Table 1. We handled “I do not know” responses separately. We then aggregated responses in two ways: by participant and by task. When aggregating by tasks, we can explore whether there are differences between how participants perform for refactorings against non-refactorings. Aggregating by participant, we can explore whether demographic information predicts performance through statistical regression.

4 RESULTS

Here, we present the study results. For RQ₁, we cover the challenges that participants reported in their review and refactoring experiences, for both this activity and in general. For RQ₂, we quantify how students performed when identifying the behavioral impacts of proposed code changes.

4.1 RQ₁: Challenges in Comparison

Our first research question focuses on the specific challenges that students face during code comparison, considering interviews, code change comprehension tasks, and reflection surveys. Through card sorting described in Section 3.7, we identified 13 individual barriers, which we sorted into the four larger categories in Table 3.

In discussing barriers henceforth, we print categories as **bolded** and individual barriers *italicized*. For example, in the category of **Tool Barriers**, there were three individual barriers, *lack of tests*, *limited or misaligned view*, and *toolchain issues*. In the interviews,

8 of the 29 participants (28%) gave responses that suggested *limited or misaligned views* is a barrier for code comprehension. This barrier came up for 17 of the 44 participants in the class activity (39%) and seven of the 44 participants in the post-activity reflection (16%). All categories were nearly equal in being evoked throughout the interviews (either 12 or 13 out of 29 per each), but the **Code Barriers** dominate the comment themes in the class activity and its reflections. For each quote from a participant, we denote the source (activity, interview, or reflection) with a subscript—i.e., P44_A means the quote was from the activity, P20_R comes from the reflection survey, and P23_I comes from an interview.

4.1.1 Context Barriers. We use “context” to mean the circumstances outside of the code that determine a developers’ relationship and responsibilities to the code. Here, three difficulties emerged: *Lack of Time*, *Social Friction*, and *Self-Doubt*.

Context Barrier 1: Lack of Time Comprehending code takes time, and time is a limited resource. This barrier hinders students from comprehensively exploring the entirety of each code change. In our datasets, students raised this issue primarily in the class reflections, wherein ten participants there explicitly stated that the 30 minutes was insufficient to address all tasks to their satisfaction. Nevertheless, two participants discuss it in interviews, supporting that the barrier is not merely a consequence of experimental design.

Context Barrier 2: Social Friction Some students remark that review and refactoring are often team activities, and there are two primary ways conflict manifests in team contexts. First, the student needs something from someone else but is unable to acquire: clear explanations about what a file does, for example, or P31_I’s frustration with “people not doing what they’re supposed to do” in their responsibilities. Second, friction manifests by what the other party provides that the student does *not* want. For example, P43_I claims: “I would say my biggest frustration is some group members always think that they’re right.”, which is similar to the “pushback” experienced in code review by developers in industry [13]. Their teammates’ attitude thus hampers reviews by stifling the flow of information. Given that the class activity was individual, the latter form of this barrier comes exclusively from the interviews about general experience, whereas the first form, the lack of social support, was invoked in the reflections as well.

Context Barrier 3: Self-Doubt Even when a student comes to a conclusion about a code change, they may not have confidence to act effectively. Uncertainties can stem from many sources: P03_I discusses how their lack of experience with GitHub tooling raises doubts about their efficacy. Responses to the class activity cite inexperience with languages or refactoring. Nevertheless, self-doubt may impact the manner in which student developers make claims about code. Furthermore, this barrier is present in prior work on sources of confusion in code review among professional developers, attesting that this is not merely an issue of being a student [12].

4.1.2 Tool Barriers. Though developer tools intend to expedite workflows, incorporating them into a student developer’s habits can come with new problems. In this category, three barriers emerged: *Lack of Tests*, *Limited or Misaligned Views*, and *Toolchain Issues*.

Tool Barrier 1: Lack of Tests For verifying behavioral equivalence, a test suite is a valued, time-saving tool. In this study, we

intentionally left tests out as we wanted to study the students’ comprehension practices in reading code. Nevertheless, participants noticed their lack, and in four of 44 reflections, test suites & debuggers were the most desired addition to the activity.

Tool Barrier 2: Limited or Misaligned View The way the interface visualizes code changes can both help and hinder comparative comprehension. Within this barrier, we note three patterns. First, the capacity of the view may be too small. Some students express frustration when relevant information is elsewhere in the program and they must search for it, as opposed to that information being readily available in their current view. As P12_A put it for task 7, “This change was immediately more annoying to look at, given its size in the diff viewer.” This issue corresponds with the *Large Scope* barrier (described later). This barrier is also seen in the search behavior of professional programmers; they frequently use code search to assist their code comprehension during code review tasks [45].

Second, students mentioned that it became more difficult to compare code when it was not positioned well. This happened to them in some cases with large change blocks: “(P37_A) The new and old version were simply split into two large chunks and not displayed side by side.” It also affected students when functions are moved, especially between files. For Task 4, P24_A noted: “The changes were on two different files, which made it a bit trickier to compare them.”

Third, an interface’s use of highlighting can be too coarse to effectively pinpoint the code a developer should attend to. Red and green highlighting, as in Figure 2, is a tool by which GitHub focuses a developer’s attention on changes. However, some students discuss how the highlighting algorithm can be distracted by irrelevant changes. For example, P33_I states, “So it’s been a lot of time trying to figure out what logic is changed, just when you realize that there wasn’t a significant change, it was just a little formatting different.”





Tool Barrier 3: Toolchain Issues Several students discuss the challenge of coordinating multiple tools during review. For example, in the interview, P02_I talks about their experiences in reviewing code: “The process of having to change branches, having to stop the program and then restart the program if you’re testing this, it’s a little tedious.” This barrier becomes a comprehension barrier when the relevant code is hidden behind several layers of tool coordination, and answering the core question—did this behavior change?—requires switching contexts. Other toolchain pain points include pull requests, continuous integration, and poor code interactivity.

4.1.3 Code Barriers. Unlike the previous categories which emphasize the conditions that a student may experience for any code, this category concerns the difficulties rising from the code itself. Participants discussed three types of barriers: the *Large Scope* of code to review, *Unfamiliar code*, and poor code *Comprehension*.

Code Barrier 1: Large Scope Changes that impact a large volume of code may require more effort to comprehend. This was the case in our Task 7, as P27_A notes the distress: “Given that there were a huge change, at first it was overwhelming to read all of the code. For the same task, P12_A identifies a potential solution—“This [pull request] could be improved by separating smaller changes into more commits, but that doesn’t appear possible in this example.” From prior work, long and complex changes are a common barrier during code review for professional developers [12]. For example, defects that appear in files presented later in a pull request more often go undetected [18].

Table 3: Barriers reported by unique student participants in the class activities and interviews.

Totals include *unique* participants per category. For example, consider the Context Barriers. In interviews, 12 participants described social friction, 2 limited time, and 1 self-doubt. With overlap, this is 13 unique interviewees, or 45% of the sample.

	Title	Description	Interview n=29	Activity n=44	Reflection n=44
	Context Barriers				
	<i>Limited Time</i>	Insufficient time to perform the task to the developer's satisfaction.	2 (7%)	4 (9%)	10 (23%)
	<i>Social Friction</i>	Dysfunctions or a lack of response from other developers.	12 (41%)	0 (0%)	2 (5%)
	<i>Self-Doubt</i>	Difficulty because of lack of experience or lack of self-confidence.	1 (3%)	3 (7%)	1 (2%)
	All Context Barriers		13 (45%)	6 (14%)	13 (30%)
	Tool Barriers				
	<i>Lack of Tests</i>	Insufficient automatic verification of the codebase.	4 (14%)	3 (7%)	4 (9%)
	<i>Limited or Misaligned View</i>	Cannot focus on all relevant code at once; limited screen space.	8 (28%)	17 (39%)	7 (16%)
	<i>Toolchain Issues</i>	Dysfunctions in coordination of tools.	7 (24%)	1 (2%)	1 (2%)
	All Tool Barriers		13 (45%)	18 (41%)	12 (27%)
	Code Barriers				
	<i>Large Scope</i>	Large volume of code to comprehend.	8 (28%)	11 (25%)	8 (18%)
	<i>Unfamiliar Code</i>	Code is unfamiliar or uses unfamiliar features.	1 (3%)	21 (48%)	10 (23%)
	<i>Comprehension</i>	Code is difficult to understand.	8 (28%)	21 (48%)	6 (14%)
	All Code Barriers		12 (41%)	32 (73%)	23 (52%)
	Comparative Comprehension Barriers				
	<i>Unclear Motivation</i>	The developer does not know why code was written or changed.	10 (34%)	3 (7%)	2 (5%)
	<i>Deep Changes</i>	New version of code looks very different.	0 (0%)	7 (16%)	2 (5%)
	<i>Merge Conflicts</i>	Dysfunctions in deciding the authoritative versions.	3 (10%)	0 (0%)	0 (0%)
	<i>Delta Comprehension</i>	The changes between code versions are difficult to understand.	1 (3%)	15 (34%)	4 (9%)
	All Comparative Comprehension Barriers		13 (45%)	18 (41%)	8 (18%)

Code Barrier 2: Unfamiliar Code Students noted the difficulty of making accurate judgments when a change included algorithms or APIs they had not seen before. In the class activity, this happened acutely at Task 3, in which a for loop was converted to use the Java stream API. Numerous students responded that this inhibited their ability to make an accurate comparison. As P08_A says, “*This was a bit more difficult as I am not experienced in using array streams.*”

Furthermore, even though we selected iTrust v2 because students had experience with it, students were not uniformly familiar with every part of it. Students remarked on the difficulty of making decisions in functions they had never seen before. As P32_A says in the class activity: “*I did not know what the code's purpose was, so I had to look at the entire file...Only looking at the change lines was difficult.*” This barrier is similar to the *lack of familiarity with existing code* barrier that appears commonly among professional programmers during code review [12, 27].

Code Barrier 3: Comprehension Code that is awkwardly written is difficult to comprehend. This difficulty therefore interferes with their ability to evaluate behavioral equivalence. In the class activity, students noted this most often on Task 3, the consolidation of multiple large conditionals. The old code was bad (P21_A: “*The old code was kind of a pain to read*”), but some said the new code was not much better (P25_A: “*the new boolean statements were somewhat tricky to parse*”). This barrier (called *lack of knowledge or ability*) is echoed in prior work as a barrier to adoption of code review practices in educational settings [24].

4.1.4 Comparative Comprehension Barriers. Like the previous category, these barriers relate directly to code but are unique to situations with more than one version. This category occurs indirectly in the literature, such as the *Behavior* category that emerged from an exploratory industrial case study focused on understanding code changes [48]. Four barriers emerged: *Unclear Motivation* for proposing changes, *Deep Changes* to code structure, *Merge Conflicts*, and *Delta Comprehension* for understanding a change.

Comparative Comprehension Barrier 1: Unclear Motivation Students look for natural-language documentation about what features the new code has in relation to the old code. Although the descriptions in the class activity were intentionally sparse to draw focus to the code, this barrier primarily emerged through the interviews. As P23_I says, “*When I don't have documentation, that definitely slows down the process of me being able to understand and interpret what their code is doing.*” In this quote, “documentation” refers to change documentation, not merely code explanations. This barrier is also found among professional populations when the code has unclear rationale [12, 41], as the purpose of a change is among the features developers desire during code review [48].

Comparative Comprehension Barrier 2: Deep Changes By *deep changes*, we mean when a contiguous segment of code is restructured so thoroughly that there remains little semblance between the old and new versions. For some students, this was the case for Task 4, where multiple conditionals were consolidated into inline variables. Concisely put by P18_A, “*When the code is very different in format, it is harder to keep track of logic in old and new cases.*”

Comparative Comprehension Barrier 3: Merge Conflicts When discussing tools during interviews, some students raised the frustrations for *merge conflicts*. These are issues where separate proposals for code changes affect the same part of the original code, leading to ambiguity about version change to keep. These situations may bring up a cocktail of different barriers—unclear motivation and unfamiliar code in how other branches have developed—and complicate a student’s understanding of the project direction altogether.

Comparative Comprehension Barrier 4: Delta Comprehension In some cases, students remarked on the difficulty of comparison as its own cognitively demanding activity beyond understanding a single piece of code (i.e., different from Code Barrier 3: *Comprehension*). It requires coordinating multiple levels of thought: not only tracking superficial changes of the text but also how it represents its implicit logic. P34_A refers to this when responding to the difficulty of “*having to parse through the removed code to piece together the bits in the new code.*” Additionally, it requires thinking not only about what is shown but what is not shown, as P20_R reflects the difficulty of “*figuring out the difference cases covered by each version and ensuring no functionality was lost.*”

RQ₁: Comparing code effectively requires the coordination of many tools and cognitive activities, and as a result, student developers report a variety of barriers across their contexts, tools, and the comprehension of code and differing versions.

4.2 RQ₂: Accuracy of Refactoring Review

Table 1 shows the number of responses per task, the number of “Yes” responses in the *Y* column, the number of “No” responses in the *N* column, the number of “I do not know” questions in the *DK* column, and for tasks 2–7 only, the percentage of correct responses in the % *CORRECT* column. The correct response for each of tasks 2–7 is underlined (i.e., “No” is correct for Task 2, and there were 7 “No” responses, representing 17% correctness). The correctness calculation excludes blank and “I do not know” responses.

4.2.1 Aggregation by Task. Students identified true refactorings correctly between 76% and 94% of the time but correctly identified behavioral changes between 8% and 17% of the time. In other words, our participants tended to identify most changes as refactorings regardless of the ground truth. One explanation is that participants misunderstand the definition of refactoring despite our introduction. To test this explanation, we can compare *what participants said the change’s impact was* (question 2 in Section 3.3) against *whether they label it as a refactoring* (question 1 in Section 3.3). For consistency in the responses, tasks labeled as refactorings should not be described with behavioral changes, and vice versa.

Table 4 shows how often label and description match. The rows represent responses to Question 1 (omitting “I do not know”) and the columns represent responses to Question 2. Cells in red represent discrepancies. Overall, we found high agreement between the responses to the questions. Only 13 out of 284 responses (4.5%) from 10 of 44 (22.7%) unique participants had disagreement. Only one label claimed a refactoring that changes behavior—one participant correctly identifying the bug in Task 2 but calling it a refactoring anyway. Of the other 12 discrepant labels from 9 unique participants, eight reason from changes to the control flow, behavior

Table 4: Discrepancies between how students label code changes and how they describe the content of the change. Q1 and Q2 are shown in Figure 3 and discussed in Section 3.3.

	Labeled as ↓	Q2		Total
		Says Behavior Changes	Says Behavior Does Not Change	
Q1	Refactoring	1	248	249
	Non-Refactoring	23	12	35
Total		24	260	284

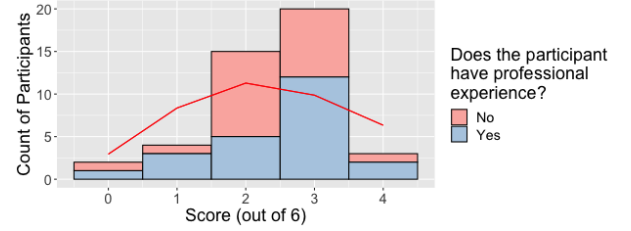


Figure 5: A histogram of individual participant scores. The probability density plot from our Conway-Maxwell-Poisson distribution ($\lambda=2.84\pm0.73$, $\nu=1.07\pm0.22$, both highly statistically significant with $p<.001$) is scaled and overlaid.

notwithstanding. For example, one participant says of Task 5, “*It moves the decision-making regarding validation from the service class to the object class.*” This statement is accurate but does not disqualify the case as a refactoring.

4.2.2 Aggregation by Participant. Figure 5 also aggregates performance by individual. For the six primary tasks, the maximum score was four and the minimum zero. The mean number of tasks correct was 2.41 (standard deviation of 0.92), and the median and mode both 3 (20 participants). The mode of 3 may reflect the expected score when overestimating behavioral similarity for all tasks.

From our demography, we hypothesized factors which explain why some participants score higher: whether the participant had experience in professional environments, and whether they used time after class to reassess and submit (true for nine of 44 submissions). We encoded both factors as binary effects. Because our performance data is a discrete value with a range between 0 and 6, we modeled this distribution with a Conway-Maxwell-Poisson distribution to account for the difference between our data’s mean and variance, which, in a basic Poisson distribution, should be equal [46]. The estimated probability mass function is overlaid onto Figure 5 and scaled according to the overall sample size. Though visual inspection suggests a slight difference in scoring rates per professional experience, neither experience nor time submitted had a statistically significant effect (for entries submitted late, $p=0.77$; for students with professional experience, $p=0.58$).

RQ₂: Per-participant data corroborate that student developers generally struggle to comprehend code changes and identify behavioral differences, with tendencies to overlook behavioral differences among syntactic differences.

5 DISCUSSION

The results in Table 3 provide additional context for the barriers observed as part of RQ₁. The combination of areas affected suggests several opportunities and actionable insights for paths forward.

5.1 Collaboration

The *Social Friction* barrier was present in 41% of the interviews, more than any other barrier in the interviews. This suggests that the collaborative aspect of code review presents some of most notable challenges for students, especially for the diversity of ways that friction manifests—from navigating fuzzy behavioral questions (“*I don’t like giving people negative feedback*” (P12_f)) to conflict negotiation (“*group members that always think they’re right*” (P43_f)) to human coordination (“*gap time between me pushing a change and the next person actually reviewing it*” (P37_f)).

Code review challenges related to social friction are not limited to student populations. *Social Friction* appears, for example, as breakdowns in *Social Interactions* in Sadowski and colleagues’ industrial case study of exclusively professional developers [44]. They highlight *tone* and *power* as two primary forms in which these barriers manifest; *tone* is resonant with some of the issues students discuss with problematic teammates, but *power* in student contexts deals with different circumstances than in professional spaces. Expanding on the concept of tone, negative sentiment emerged as a factor that decreases the usefulness of review comments in Bosu and colleague’s analysis of interactions at Microsoft [7]. In mixed populations with student, professional, and open source developers, prior work shows that toxicity and negativity are common in code review comments, and it can be particularly problematic for women in professional environments [21][38].

These observations motivate future work that can ease the social tension at all levels. For example, for bridging transitions throughout the classroom and into broader software development spaces, we can learn from Ford and colleague’s work on mentorship in online platforms [16]. Explicit instruction on tone in code review comments and other aspects of code review interactions can be essential to facilitate better inclusion over time.

5.2 Comparative Comprehension

In the class activity specifically, the dominant barriers in our framework were **Code**: in particular, *Unfamiliar Code* and *Comprehension*. This resonates with prior literature, from Bacchelli and Bird’s framing of MCR as a comprehension task in essence [4] to the compiled summary in Davila and colleagues’ systematic literature review of modern code review [11].

The results for RQ₂ corroborate that student developers struggle to identify behavioral differences in the code changes. Our prior work on comparative comprehension also examines the accuracy in identifying behavioral similarity in similar code [33]. There, the population included 68 graduate students, 17 professional software developers, and 10 people with other roles. With that more experienced population as compared to ours, they reported only 43% accuracy on identifying behavioral differences in non-clones. This result is still higher accuracy than we find for identifying behavioral changes (8% to 17% per Section 4.2), but all these results indicate correct responses in under half the instances.

We can propose explanations of this difference—the refactorings in the tasks were more complex, the population was less experienced, and the tasks were performed under time pressure, for example. Even if we adjusted the methodology to account for these explanations, however, some circumstances would persist in real-world environments to varying degrees—insecurities about familiarity, experience, and a lack of time plague more experienced developers in industry, too [12]. Nevertheless, these results raise questions about the underlying process of comparative code comprehension.

The low correctness rate on non-refactorings specifically could also be a form of the anchoring effect. The anchoring effect is a form of cognitive bias in which a person tends toward answers that are close to some provided starting point for their inquiry [19]. It has been observed elsewhere in software engineering, such as in how displaying metrics of code comprehensibility influences developers’ own assessments of it [51]. In our case, participants may start the task primed to think in terms of similarity rather than dissimilarity, using behavioral identity as a default hypothesis and responding with “no” only if they can discover evidence.

For the barriers that exist because of how developers are able to interact with code, scaffolding to assist comparative comprehension may help student developers, and it could take a number of forms. One form could be behavioral differencing. For example, some prior work explicitly reveals the program state and show developers which variables change for a particular test [14]. Another approach to combat the *Large Scope* barrier directly could be decomposing pull requests into small pull requests, each containing atomic changes. Prior work has also identified this as a need [5, 6, 49]. For example, Tao and Kim analyzed 453 code revisions in OSS projects to deduce that 17% of that sample mixes multiple changes into a single pack, and they reduced the burden on reviewers by slicing these composite changes into smaller cohesive ones [49].

Yet another approach could be to add comprehension information to both the old and new versions of code. For example, code summarization could be added to the pull request interface, such as that explored by Buse and Weimar for program changes [8], to combat issues related to *Limited or Misaligned Views*, comprehending changes in *Unfamiliar code*, or comprehending changes with *Large Scope*. Refactoring-aware review [20] or semantic code clone detection [30] can be deployed to reduce the cognitive work required of developers to compare code. In essence, automated semantic annotations that explain the context of a code change would substantially reduce the cognitive load of comparative comprehension. While it is not clear how much of this scaffolding would be retained as students gain more experience in their careers, the immediate frustrations could be alleviated in the meantime.

5.3 Comparison is a Process in Context

From our discussions with participants, we find that “comparison” is not monolithic but a process comprising smaller activities. In the practice of code review, comparison is not merely having two algorithms neatly presented and free of noise. Rather, it involves noticing differences, navigating between them, and making decisions from them. This observation resonates with literature of how students learn from comparing and analogizing between concepts in general. For example, Alfieri and colleagues survey literature on

how students may learn and transfer new concepts through analogizing, pointing specifically to constituent activities like searching and aligning concepts [1]. Patitsas and colleagues apply the premise explicitly to computer science by experimenting with presenting alternative algorithms, seeing learning benefits in students who studied algorithms side-by-side rather than sequentially [37].

In our case, we can envision elements of comparison that may contribute to a developer approving or rejecting a proposed code change, such as the following:

- *Establishing Syntactic Difference*: The developer perceives where differences occur in the text or style of code.
- *Comprehending...*
 - *...One Alternative*: The developer manually investigates what each version does independently of the other.
 - *...Semantic Differences*: The developer reasons about what each version does to build up to what they may do differently.
- *Mapping Corresponding Semantic Features*: Under the assumption of behavioral equivalence, the developer associates elements of code to compare. If the assumption does not exist, this activity is de-emphasized.

Decomposing a process into steps like these also helps us identify opportunities for intervention at specific moments. Some existing tools in the GitHub interface reflect steps in the process—difference highlighting, for example, focuses on expediting *establishing syntactic difference* by narrowing the amount of code developers must inspect before finding the divergence. Test suites, likewise, offload the cognitive work of *comprehending one alternative* or *differences* by automating behavior verification and communicating intended code behavior. Thinking forward in this process can motivate research and industry to augment the semantics-related elements of the process.

6 THREATS TO VALIDITY

In this section, we discuss the weaknesses in our research. We organize them into three categories: construct threats for the concepts we used, internal threats in how we enacted methods and interpreted findings, and external threats in how our findings generalize to the broader landscape of software engineering.

Construct Validity. First, we operationalize code change comprehension through yes-or-no questions about behavior, but this has limitations. A binary response for a complex phenomenon like comprehension may give false signals that a student understands the change when answering correctly by chance. Developers envision code comprehension as a spectrum, and tasks can be completed successfully at different levels of comprehension [4]. Therefore, future work could focus on more expansive observations of developer behavior to measure comparative comprehension.

Second, code review and refactoring have established definitions in the software engineering community but can vary in practice. Hence, accurately identifying something as a refactoring depends on what you accept as the definition. For a classroom study nevertheless, the concise definitions of refactoring—semantics-preserving changes—have value for introducing the practice.

Internal Validity All of our sources of data—interviews, task reflections, and post-activity surveys—are approximations of a student’s challenges. In interviews, the students may misunderstand

the topic, although we mitigated this threat by asking them to define their understanding of our concepts “review” and “refactoring” so we could adjust appropriately. Additionally, they are drawing on their memories extemporaneously, and they may not be able to recall all relevant experiences during the interview itself.

The class activity was time-limited because the lecture period lasts only 75 minutes. As a result, participants may have felt pressured to submit responses before they wanted. This constraint may explain the results if students begin with a hypothesis that the changes are refactorings, and it takes effort to discover evidence against it. It may also limit the depth of reflection in their free responses. To mitigate this, we allowed students to submit the form later that evening if they were motivated to continue work, and we tested the differences in correctness to account for this choice.

Additionally, student participants may be influenced by power dynamics when they report experiences related to the class. We attempted to mitigate this threat by promising anonymity and reassured them their grade would not be impacted by their participation (beyond the extra credit for interviewees) or lack of participation.

External Validity. The code change comprehension tasks were situated with a specific set of refactorings, in a specific language, in a specific code base. The results may not generalize beyond this context. For one, the particular refactorings that we implemented here may not sufficiently capture the variety in practice, and the barriers discoverable in the data could be conditioned by the refactoring types that produced the data. Future research should explore the different impact of different types of changes—from renames to algorithmic rewriting—with more variety.

Our population—a single class in a single semester at a single university—may not generalize. Student efficacy in these tasks may be a product of their training. As such, curricula that emphasize review in lectures may be more or less sensitive to refactorings in tasks like these. Even accounting for these contexts, this study differs from its immediately surrounding course context in that these pull requests do not impact their class project.

7 CONCLUSION

In this paper, we showed qualitative and quantitative evidence that students face a broad set of barriers when comparing code changes. We report that for students, comparing code requires coordination of cognitive activities, as well as tools, in a way that is challenging and often frustrating. Furthermore, given that many of the challenges that students face, such as *unclear motivation* or *unfamiliar code*, persist into professional environments [48] this study increases the urgency with which issues in change-based or comparative comprehension should be addressed.

We also confirm the findings of prior work that participants tend to overestimate behavioral equivalence when assessing proposed code changes [33]. It would seem that code review is sufficiently cognitively demanding, and requires sufficient coordination among tools, that is challenging for developers regardless of expertise.

8 ACKNOWLEDGEMENT

This work is funded in part by NSF SHF #2006947 and #1749936. Thank you to Drs. Kai Presler-Marshall, Alexandra Milliken, and Souti Chattopadhyay for help in the study execution and analysis.

REFERENCES

- [1] Louis Alfieri, Timothy J Nokes-Malach, and Christian D Schunn. 2013. Learning through case comparisons: A meta-analytic review. *Educational Psychologist* 48, 2 (2013), 87–113.
- [2] Eman Abdullah AlOmar, Hussein Alrubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25–28, 2021*. IEEE, 348–357. DOI: <http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00044>
- [3] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23–24, 2022*. ACM, 689–701. DOI: <http://dx.doi.org/10.1145/3524842.3527932>
- [4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.
- [5] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1*. IEEE, 134–144.
- [6] Tobias Baum and Kurt Schneider. 2016. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22–24, 2016, Proceedings 17*. Springer, 301–308.
- [7] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 146–156.
- [8] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the 25th IEEE/ACM international conference on automated software engineering*. 33–42.
- [9] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265.
- [10] Chun Yong Chong, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2021. Assessing the students' understanding and their mistakes in code review checklists: an experience report of 1,791 code review checklist questions from 394 students. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 20–29.
- [11] Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software* 177 (2021), 110951.
- [12] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 49–60. DOI: <http://dx.doi.org/10.1109/SANER.2019.8668024>
- [13] Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspán, and James Lin. 2020. Predicting Developers' Negative Feelings about Code Review. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 174–185.
- [14] Khashayar Etemadi, Aman Sharma, Fernanda Madeiral, and Martin Monperrus. 2023. Augmenting Diff's With Runtime Information. *IEEE Transactions on Software Engineering* (2023), 1–20. DOI: <http://dx.doi.org/10.1109/TSE.2023.3324258>
- [15] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hakenberg. 2012. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 73–82. DOI: <http://dx.doi.org/10.1109/ICPC.2012.6240511>
- [16] Denae Ford, Kristina Lustig, Jeremy Banks, and Chris Parnin. 2018. "We Don't Do That Here" How Collaborative Editing with Mentors Improves Engagement in Social Q&A Communities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–12.
- [17] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- [18] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalkılı, and Alberto Bacchelli. 2022. First Come First Served: The Impact of File Position on Code Review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 483–494. DOI: <http://dx.doi.org/10.1145/3540250.3549177>
- [19] Adrian Furnham and Hua Chu Boo. 2011. A literature review of the anchoring effect. *The journal of socio-economics* 40, 1 (2011), 35–42.
- [20] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 71–79.
- [21] Sanuri Dananja Gunawardena, Peter Devine, Isabelle Beaumont, Lola Piper Garden, Emerson Murphy-Hill, and Kelly Blincoe. 2022. Destructive Criticism in Software Code Review Impacts Inclusion. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 292 (nov 2022), 29 pages. DOI: <http://dx.doi.org/10.1145/3555183>
- [22] Sarah Heckman, Kathryn T. Stolee, and Christopher Parnin. 2018. 10+ Years of Teaching Software Engineering with ITrust: The Good, the Bad, and the Ugly. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 1–4. DOI: <http://dx.doi.org/10.1145/3183377.3183393>
- [23] Bradford W. Hesse. 1995. Curb cuts in the virtual community: telework and persons with disabilities. In *28th Annual Hawaii International Conference on System Sciences (HICSS-28)*, January 3–6, 1995, Kihei, Maui, Hawaii, USA. IEEE Computer Society, 418–425. DOI: <http://dx.doi.org/10.1109/HICSS.1995.375707>
- [24] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of peer code review in higher education. *ACM Transactions on Computing Education (TOCE)* 20, 3 (2020), 1–25.
- [25] Sarah Jessup, Sasha M Willis, Gene Alarcon, and Michael Lee. 2021. Using eye-tracking data to compare differences in code comprehension and code perceptions between expert and novice programmers. (2021).
- [26] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.
- [27] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *Proceedings of the 38th international conference on software engineering*. 1028–1038.
- [28] SeolHwa Lee, Andrew Matteson, Dania Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. 2016. Comparing programming language comprehension between novice and expert programmers using eeg analysis. In *2016 IEEE 16th international conference on bioinformatics and bioengineering (BIBE)*. IEEE, 350–355.
- [29] Moira Maguire and Brid Delahunt. 2017. Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars. *All Ireland Journal of Higher Education* 9, 3 (2017).
- [30] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. SLACC: Simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 210–221.
- [31] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
- [32] Andrew Meneely, Ben Smith, and Laurie Williams. 2012. Appendix B: iTrust electronic health care system case study. *Software and Systems Traceability* (2012), 425.
- [33] Justin Middleton and Kathryn T Stolee. 2022. Understanding Similar Code through Comparative Comprehension. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–11.
- [34] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. DOI: <http://dx.doi.org/10.1109/TSE.2011.41>
- [35] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 125–136.
- [36] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 135 (nov 2018), 27 pages. DOI: <http://dx.doi.org/10.1145/3274404>
- [37] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. 2013. Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 145–152.
- [38] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. 2019. Expressions of Sentiments during Code Reviews: Male vs. Female. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 26–37. DOI: <http://dx.doi.org/10.1109/SANER.2019.8667987>
- [39] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. 2017. Investigating eye movements in natural language and c++ source code-a replication experiment. In *Augmented Cognition. Neurocognition and Machine Learning: 11th International Conference, AC 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9–14, 2017, Proceedings, Part I 11*. Springer, 206–218.
- [40] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. 2022. Correlates of programmer efficacy and their link to experience: A combined EEG and eye-tracking study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 120–131.
- [41] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 202–212.
- [42] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. 2014.

- Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–33.
- [43] Christian Rohrer. 2014. When to use which user-experience research methods. *Nielsen Norman Group* 12 (2014).
 - [44] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.
 - [45] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 191–201. DOI : <http://dx.doi.org/10.1145/2786805.2786855>
 - [46] Galit Shmueli, Thomas P Minka, Joseph B Kadane, Sharad Borle, and Peter Boatwright. 2005. A useful distribution for fitting discrete data: revival of the Conway–Maxwell–Poisson distribution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 54, 1 (2005), 127–142.
 - [47] Xiangyu Song, Seth Copen Goldstein, and Majd Sakr. 2020. Using Peer Code Review as an Educational Tool. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 173–179. DOI : <http://dx.doi.org/Song2020Peer>
 - [48] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How Do Software Engineers Understand Code Changes? An Exploratory Study in Industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 51, 11 pages. DOI : <http://dx.doi.org/10.1145/2393596.2393656>
 - [49] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 180–190.
 - [50] Deborah A. Trytten. 2005. A Design for Team Peer Code Review. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*. Association for Computing Machinery, New York, NY, USA, 455–459. DOI : <http://dx.doi.org/10.1145/1047344.1047492>
 - [51] Marvin Wyrich, Andreas Preikschat, Daniel Graziotin, and Stefan Wagner. 2021. The mind is a powerful place: How showing code comprehensibility metrics influences code understanding. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 512–523.