



Understanding Comparative Comprehension Barriers for Students during Code Review through Simplification

Nick Case 
North Carolina State University
United States
nickrcase@gmail.com

John-Paul Ore 
North Carolina State University
United States
jwore@ncsu.edu

Kathryn T. Stolee 
North Carolina State University
United States
ktstolee@ncsu.edu

Abstract

Comparative code comprehension – the cognitive activity of understanding how two pieces of code behave relative to each other – occurs during code review, code search, and debugging. Recent research revealed barriers that prevent students from effectively practicing comparative code comprehension during code review and determining whether a code change is behavior-preserving. This prior work suggests it is a difficult task for students, with an accuracy of 44%. However, the tasks in prior work had high complexity, potentially leading to artificially low success rates for students. In this work, we also study comparative code comprehension with students but with simplified tasks in an effort to reduce the previously identified barriers and set students up for high levels of success. We conjecture that simplification ensures that the failures and barriers students encounter are as foundational as possible. Specifically, we target barriers within the tool, code, and comparative comprehension categories from prior work. We conduct an empirical study with 125 students and eight tasks of varying sizes with code changes of varying complexity. We discover that with the chosen experimental setup, students could successfully recognize changes in code behavior with an accuracy of 89%. When students made mistakes, these fell into two primary categories: misunderstanding of behavior preservation and shallow or incomplete evaluation of change logic. Our results can benefit software engineering educators seeking to incorporate code review into the classroom while minimizing barriers.




CCS Concepts

• **Software and its engineering** → **Development frameworks and environments**; **Software development techniques**; **Software creation and management**.

Keywords

Code Review, Code Comprehension, Human Factors

ACM Reference Format:

Nick Case , John-Paul Ore , and Kathryn T. Stolee . 2025. Understanding Comparative Comprehension Barriers for Students during Code Review through Simplification. In *33rd ACM International Conference on*

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.

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1276-0/2025/06

<https://doi.org/10.1145/3696630.3727249>

the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3696630.3727249>

1 Introduction

Developers review code, restructure code, and choose between alternative implementations to improve code quality [5], readability [39], and to spread knowledge throughout a team both in professional [33] and open-source contexts [34]. These tasks all depend on a developer's ability to compare and comprehend what has changed between different versions of code, a kind of comparative comprehension [28]. Because comparative comprehension is an essential and non-trivial skill for developers, many computer science (CS) and software engineering (SE) curricula have begun emphasizing code review processes [18]. Beyond code review tools themselves, like GitHub, students face numerous barriers in comparative comprehension tasks and struggle to accurately distinguish between a behavior-preserving change and a non-equivalent change [28], a challenge shared with professional developers [3].

Prior work in comparative comprehension with students reveals a set of barriers that students face when determining whether or not a code change is behavior-changing [26]. These barriers are related to the code itself, the tools, the context of performing the task, and the difficulty of performing the comparison. However, in the prior study, some of the more dominant barriers, specifically those related to comprehending and reviewing large changes, were likely the result of the researchers' chosen experimental tasks.

This work seeks to assess the accuracy with which students perform a code review of behavior-preserving changes using an experimental setup that intentionally simplifies many of the barriers identified in the prior work. Our approach adapts the concept of *simplification* [46] from Linguistics, which seeks to improve teaching and learning of languages by focusing on reducing complexity, removing barriers, and identifying the foundational shortcomings that learners are missing. If removing barriers does not improve accuracy, it is likely that the code review task is harder than we thought. If students can accurately distinguish between a behavior-preserving change and a non-behavior-preserving change, then the remaining mistakes may reflect foundational misunderstandings. By calibrating for a very high task success rate through simplification, we hypothesize that the remaining failures and barriers are as foundational as possible.

To test our conjecture, we ran a study in a third-year undergraduate software engineering class at our university. The students were instructed to determine if a code change preserved the behavior of the original code in terms of inputs and outputs. We selected program pairs that are known to be either equivalent or non-equivalent

from the validated EqBench dataset [6] and presented them to students as Pull Requests (PRs) on GitHub. For each task (i.e. PR), students reviewed proposed changes that were either equivalent (the code change does not change code behavior) or non-equivalent (the code change does change code behavior). Students then either accepted and merged the changed code if they believed the code change was an equivalent change or explained why it was not a behavior-preserving change in the PR comments. This provides us with data to understand the accuracy of students on these tasks and the reasoning behind why students made mistakes based on their comments. Our contributions in this work include:

- (1) Evidence that, in some contexts, students recognize behavior-preserving changes with high accuracy (89%), in contrast to results of prior work;
- (2) Classification of the mistakes students made after simplification, showing the most foundational mistakes are misunderstandings about behavior-preserving changes and incorrect reasoning about code changes;
- (3) Identification of a new Context Comprehension barrier for when students struggle to understand the impact of a code change because they lack familiarity with how the code is being used in practice; and
- (4) Suggestions for SE educators on how to improve teaching code change comprehension.

2 Related Work

This work is most closely related to prior work in code review in education and comparative code comprehension.

2.1 Code Review in Education

Research in software engineering education has increasingly focused on code review as a pedagogical tool [20], in part because of the importance of Modern Code Review in practice [37]. Chong *et al.* [9] examined how having students build a code review checklist could benefit the review process, and like our work, they found that simplifying code review by breaking it into steps was beneficial for students. Unlike their work, we sought to reduce the complexity of the code review task before students undertook the code review task.

Hundhausen *et al.* [19] investigated how *pedagogical code review*, an adaptation of formal code inspection, can improve students' proficiency in code review and code comprehension. Similarly, we aimed to improve the students' proficiency in code review through a novel methodology; however, we targeted the simplification of code change barriers.

A key difference between prior work about code review in education and this study is that in prior work, students are focused on detecting defects within the code [29, 42]. On the other hand, code review in this study focuses on students detecting *behavior-preserving changes*, similar to refactoring review [32].

Song *et al.* [41] built a tool specifically to aid and track peer code review that students found useful. Other tools, such as games, chatbots, and Generative AI-integrated tools, have additionally been found to help teach and aid students' peer code review (e.g. [4, 15, 31]) Like their work, we are interested in code review in education,

but unlike their work, we focus on using an industry-standard code review tool (the GitHub PR interface).

2.2 Comparative Code Comprehension

Comparative code comprehension is relatively new to the literature, with only a few papers on the topic [8, 26–28]. While code review itself is well-represented in the literature beyond its role in education (e.g., [5, 7, 11, 22, 23]), code review represents just one of the contexts for comparative comprehension. Code comprehension research is well-established as well (e.g., [13, 30, 47]), yet the focus is usually on understanding a single piece of code rather than a change to code behavior. Here, we summarize findings in comparative code comprehension.

Comparative comprehension happens in a variety of contexts. Through interviews, researchers found comparative comprehension occurs in refactoring and code review contexts [28]. Additionally, comparative comprehension was observed when participants compared different versions of code during mutation testing tasks [8]. We focus on code review in GitHub, similar to the refactoring study [26]. Researchers previously measured the accuracy with which student and professional developers identify code clones [28] or identify refactorings in a student context [26]. Accuracy for detecting type-IV [36] clones was 65% [28] and accuracy for detecting refactorings was 44% [26].

Previous work on understanding how developers engage with and understand code changes emphasized the centrality of “change understanding” to the Software Engineering process [43]. AlOmar *et al.* [2] identified that a particular challenge during refactoring review is distinguishing between equivalent and non-equivalent versions. The tasks in this study required identifying equivalent and non-equivalent changes; however, these code changes were unassociated with code smells and refactorings [48]. Studying behavior-preserving changes allows for the cognitive process of comparative comprehension to be studied and then applied to other fields of program comprehension [21].

The structure of the code may have an impact on code comprehension, and several of the tasks in our study involved modifications to control flow structures (e.g., Figure 1). Some prior work finds that changes in the structure of API code examples impacts the time to make a judgment about their understanding but does not impact the accuracy of understanding [1]. However, considering the comparison of two examples in isolation, prior work finds that changes in structure for code snippets has a significant impact on their comparative code comprehension [28]. While we do not specifically measure and study the impact of such control structure changes on participant accuracy, the specific example of Task E in Figure 1, which involves turning nested if-statements into a single if-else-statement and thereby extensive structural changes, had the highest accuracy per Table 1. It would seem, then, that larger structural changes, do not always impede comprehension.

3 Experimental Methodology

Our goal is to understand more about the challenges students encounter when deciding if a code change is behavior-preserving. We arrange the research around three questions that measure: task accuracy, the types of mistakes made, and which of the previous



Figure 1: The GitHub PR User Interface for the equivalent version of Task E (sign).

barriers to comparative comprehension remain in this simplified experimental design. Specifically, students were asked to decide whether “code changes preserve the behavior of the original code”. Within this study, behavior-preserving means “a change was made to the internal structure of software without changing its observable behavior”. Refactoring is a common example of this; however, the tasks within this study are not refactorings due to their lack of code smells. Our study answers the following research questions:

RQ1 : How accurately do student developers recognize behavioral changes in code review tasks?

This question is important because we do not know if students struggle with code review tasks and the associated comparative comprehension because of the *inherent* task difficulty or because of *adjacent* barriers that can be mitigated separately from the core comprehension task. To explore this, we looked at how students performed on eight code review tasks, presented in either equivalent or non-equivalent pairs. Using the GitHub Pull Request interface and other optional resources like Generative AI, web search, and StackOverflow, students either accepted the changes as behavior-preserving or explained a behavior change using GitHub comments.

Even though we are setting up students to succeed by simplifying the tasks, we know that there will still be mistakes and misunderstandings, which leads to our second research question:

RQ2 : What mistakes do students make when identifying changes in behavior?

We use the PR comments to classify the student mistakes. We approached improving code review education through the simplification of comparative code comprehension; however, an important question to consider is how successful we were in that process.

RQ3 : How effective was the process of simplification to study comparative comprehension?

We approach this research question by asking participants an open-ended question about what was *difficult* about the code review task question 3b of the post-activity survey (Figure 2).

3.1 Study Design

Our empirical study was designed with eight code review tasks based on the EqBench [6] dataset, put in GitHub PRs, and given to students for assessment. To facilitate analysis, we collected participant interactions with GitHub using the GitHub API as well as a post-activity survey.

3.1.1 Artifact Selection. The EqBench dataset was chosen for three main reasons: 1) formal verification ensured that our tasks had a ground truth, 2) it contained Java programs which is the primary programming language used by our student population and 3) the tasks in the dataset represent a variety of programs with different characteristics including length, complexity, and use of outside libraries. These characteristics matched with the barriers we aimed to simplify, giving us a pool of programs to pick from. For example, EqBench has programs smaller than 10 LoC and programs greater than 60 LoC. This allowed us to address the barrier of *Large Scope* without having to artificially reduce or expand program files (see Section 3.1.4). Additionally, each original program is accompanied by both equivalent and non-equivalent versions of it.

Table 1 shows a summary of the eight programs used in this study. To obtain this set, we classified each program in EqBench based on the size of the original code and the depth of the change. For size, we used cloc [10] to find the mean lines of code for EqBench. A program with fewer than 27 LOC was classified as *small*, otherwise it was classified as *large*. For depth, the EqBench dataset comes with metadata containing the percent of statements changed (churn). The median churn among all programs was 10%, and we used that threshold to classify the depth of a program as *shallow* (lower) or *deep* (higher). Once the dataset was classified by size and depth, we chose two tasks from each combination of size and depth. In Table 1, the *Task* column identified each task by letter, which we use throughout this paper. The program name from the EqBench dataset is in the *Program* column, followed by the characteristics in terms of size and depth. *LOC* specifies the size and *Changed Statements (%)* specifies the depth. The final three columns are the task accuracy, as reported in Section 4.1.

3.1.2 Experimental Task Design. The experimental task design is a balanced block treatment [12], illustrated in Table 2. Each task (A-H) had two treatments: an equivalent treatment (Eq) and a non-equivalent treatment (Neq) from the EqBench dataset. Each block contains all eight tasks. Treatments were assigned to tasks within a block using a Latin rectangle [12] with balanced columns. We used a Latin rectangle because we enforced an additional constraint where each participant received four equivalent and four non-equivalent tasks. This was balanced because prior work showed students had much higher accuracy on equivalent tasks compared to non-equivalent tasks. [26] Additionally, for each combination of {Size, Depth}, each participant had one equivalent treatment and one non-equivalent treatment. That is, if A (*Small, Shallow*) was equivalent, then B (*Small, Shallow*) was non-equivalent. This

Table 1: Experimental tasks from EqBench, with characteristics and task accuracy computed from the study.

TASK	PROGRAM	CHARACTERISTICS {SIZE, DEPTH}	SIZE (LOC)	DEPTH (% CHURN)	ACCURACY EQUIV (%)	ACCURACY NON-EQUIV (%)	STAT. SIG.	ACCURACY TOTAL (%)
A	pythag	Small, Shallow	21	9%	89%	91%		92%
B	limit1	Small, Shallow	14	9%	49%	90%	**	70%
C	pow	Large, Shallow	28	4%	90%	94%		92%
D	conflict	Large, Shallow	35	6%	-	-		-
E	sign	Small, Deep	15	37%	98%	91%		95%
F	optimization	Small, Deep	18	29%	81%	95%	*	88%
G	tcas	Large, Deep	191	10%	91%	97%		94%
H	ejhash	Large, Deep	48	28%	80%	89%		87%
TOTALS					82%	93%	***	89%

Statistical significance compares equivalent and non-equivalent tasks; ****: $p < 0.001$ (highly significant), ***: $0.001 < p < 0.01$ (very significant), **: $0.01 < p < 0.05$ (significant).

Problem D is excluded from accuracy results due to a processing error (see Section 4.1).

Table 2: Balanced block design showing assigned treatments to tasks: Equivalent (“Eq”) & Non-equivalent (“Neq”).

ID	A	B	C	D	E	F	G	H
0	Eq	Neq	Eq	Neq	Eq	Neq	Eq	Neq
1	Neq	Eq	Eq	Neq	Eq	Neq	Eq	Neq
				⋮				
7	Neq	Eq	Neq	Eq	Neq	Eq	Eq	Neq
8	Eq	Neq	Eq	Neq	Eq	Neq	Neq	Eq
				⋮				
13	Neq	Eq	Eq	Neq	Neq	Eq	Neq	Eq
14	Eq	Neq	Neq	Eq	Neq	Eq	Neq	Eq
15	Neq	Eq	Neq	Eq	Neq	Eq	Neq	Eq

yielded 16 blocks. In Table 2, for example, block 0 has Task A with equivalent programs and Task B with non-equivalent programs.

Participants are assigned to blocks uniformly at random. The assigned block was not revealed to the subjects. Once the block was assigned, the tasks A–H were randomly ordered per participant and added as PRs to a GitHub repository assigned to each participant. The randomization of task order was intended to avoid learning effects. That is, problem A may appear as PR#1 to one participant but as PR#4 for another; we test for learning effects as part of RQ1.

3.1.3 Study Activities. For each PR (as shown in Figure 1) in their repository, the subjects were instructed to review the change and decide if the change was behavior-preserving. They were given the following instructions in the repo’s README:

Your job is to review each Pull Request ... The intention of the review is to decide if the code changes preserve the behavior of the original code. That is, before and after the change, the code should always behave the same. Your review will result in one of two outcomes:

- 1) Approval of change & then Changes merged (if the code behavior is preserved)*
- 2) Request Changes (explain why) & then Pull Request closed (if the code behavior is not preserved)*

Because this population of students was experienced with code review in GitHub already (see Section 3.5), expecting students to review PRs and leave comments was a task consistent with their in-class experiences. Additionally, framing this study within the context of GitHub Pull Requests adds real-world context for the

process of code review. Prior work used a Google Form to provide feedback [26], but we felt that would create unnecessary context switching, potentially interfering with the study process.

3.1.4 Study Design to Simplify Barriers. Real-world code review tasks are often complex, with many developers working together through a variety of tools [38]. The barriers identified through previous work reflect the complex context that developers navigate in code review. To focus on the part of code review related to comparative comprehension, we simplify barriers to comparative comprehension that were most prevalent in the in-class code review activity in prior work [26]. We specifically targeted the barriers that appeared for at least 5% of the participants. Here, we briefly describe the barrier, specify its prevalence (as the % of participants who expressed that barrier) in the prior work, and explain how it was simplified in this work:

Unfamiliar Code – 48%. In this barrier, “code is unfamiliar or uses unfamiliar features.” [26]. When we selected the tasks for this study, we intentionally chose programs that contained familiar and basic language features and standard libraries. For example, some of the programs within EqBench contain mathematical functions such as cosine and sine, which are familiar to participants in our study because they are required to take a Calculus course. We avoided programs with unfamiliar concepts or functions.

Comprehension – 48%. In this barrier, “code that is awkwardly written is difficult to comprehend” [26]. We chose programs similar to what students had seen before in their computer science classes. We scrutinized changes to control flow, program structure, and library usage to ensure that students would have an easier time understanding the tasks.

Limited or Misaligned View – 39%. In this barrier, “the way the interface visualizes code changes can both help and hinder comparative comprehension” [26]. Figure 1 shows the view of a PR as seen by participants. We chose programs that could easily be viewed as a PR with minimal or no scrolling on a standard laptop screen. Then, immediately before the study during an in-class demo, we demonstrated how to switch between unified or split views.

Delta Comprehension – 34%. In this barrier, “students remarked on the difficulty of comparison as its own cognitively demanding activity beyond understanding a single piece of code” [26]. This barrier is not one we want to simplify because comparative comprehension is

precisely what we want to study. *Delta Comprehension* represents the core of comparing changes between code versions and figuring out if they are equivalent or non-equivalent.

Large Scope – 25%. In this barrier, “changes that impact a large volume of code may require more effort to comprehend” [26]. We chose tasks that were contained in one file. While some tasks are characterized as “Large”, our range for large programs was 28–191 LOC (Table 1). The largest code change of any task in our study was 19 LOC (Task G), less than half the median LOC changed per PR (42 LOC) found in a large-scale analysis on GitHub [40]. This places our changes as “small” compared to real changes.

Deep Changes – 16%. In this barrier, “when a contiguous segment of code is restructured so thoroughly that there remains little semblance between the old and new versions” [26]. *Deep Changes* in a task have more lines changed compared to their previous version and appear to be noticeably different. Shallow tasks are the opposite, looking very similar to their previous version. Having both shallow and deep tasks allows for this study not only to simplify the barrier but to also study the effects of shallow versus deep changes.

Unclear Motivation – 7%. In this barrier, “the developer does not know why code was written or changed” [26]. We added a motivation for these changes within the commit message for each task. Each task had the motivation of “Update code to adhere to code standards”, providing a basic motivation for why code review is necessary. This is important as developers’ code comprehension is significantly influenced by scenarios [14].

Unaddressed Barriers. Some barriers in the prior work impacted more than 5% of the participants but we were not able to address them. For example, *Limited Time* impacted 9% of participants, but to maximize the number of participants, we retained this barrier from the previous study design because this study was also executed in a classroom environment (Section 3.3). *Lack of Tests* impacted 7% of participants, but due to using the EqBench dataset for the tasks, there were no built-in tests. *Self Doubt* impacted 7% of participants, but our study focuses on the comparative comprehension component, not students’ confidence. Future studies could focus on providing participants with information to improve their confidence in code review tasks.

3.2 Study Analysis

We applied the following metrics and procedures to analyze our collected data.

3.2.1 RQ1. Task accuracy was evaluated on a binomial scale since the student was either correct or incorrect in their assessment of equivalence. We hypothesize that the non-equivalent task is easier than the equivalent task because determining non-equivalence only requires finding a single concrete difference in the external behavior of the code (“there exists a difference”). In contrast, the equivalent task requires students to convince themselves that no difference in external behavior exists (“for all cases, no difference exists”). However, in the prior work, students tended to over-estimate equivalence leading to the opposite situation [26].

We look for the impact of program characteristics and participant characteristics on accuracy, as well as the impact of behaviors

during the study, such as the use of AI-based tools (like ChatGPT or Copilot). The characteristics of the participants and their use of tools were solicited through the survey instrument shown in Figure 2, specifically questions 1, 2, and 3a.

To test how these experimental factors affect task accuracy, we created a binomial linear model with the individual participant as a random effect along with contrasts to test relationships in the data. We used the following formula: $accuracy \sim problem + order + 1/person$ to test the dataset. That is to say, the accuracy is based on three factors: the *problem* itself (Tasks A–H), the *problem order* (which task appeared first, second, and so forth), and a random effect related to the *person* performing the task. In consultation with a statistician at North Carolina State University, we chose a binomial linear model with one random effect variable as it allows us to see how accuracy (the binary response variable) can be formed as a function of the different predictor variables (problem and order). In addition, we used contrasts *within* the linear model to test the relationships even further, allowing us to observe the impact of each predictor variable and against each other to see effects on the response variable (accuracy). We also used the Mann-Whitney U test [24] (the `wilcox.test()` function in R) to compare populations based on certain characteristics within the participant dataset.

3.2.2 RQ2. When students detect a difference in the code behavior, they add a comment on GitHub explaining the difference. To answer this research question, we look at cases when students received an equivalent pair of code snippets, but claimed there was a difference. We manually categorize their inline code comments in the GitHub PR interface as well as their overall comments to determine what mistakes they made; two researchers discussed each case and concluded the root cause. Note that we focus on when students erroneously conclude that a program pair is non-equivalent because we were unable to see the opposite case—when students thought the programs were equivalent but there was an observable difference—as no comments were required per the experimental setup.

3.2.3 RQ3. With these responses, we perform a closed card sort [35] using the barriers identified in prior work as the predefined categories [26]. This analysis allows participants to reflect on their experiences without biasing their responses, which could happen if we used, for example, multiple-choice questions. Moreover, doing a closed card sort allows us to directly map the barriers identified in this study to those previously identified.

Following best practices [35] for card sorting, two authors sorted responses into categories independently, and upon completion, the two authors reviewed how each response was categorized. Responses can belong to more than one category since participants could describe multiple barriers.

3.3 Study Execution

Our study was conducted near the end of the semester, after students were accustomed to using the GitHub PR mechanism for the class project. The students participated in the study as part of an optional in-class activity that gave them 3% extra credit on their final exam (1% of the course grade). All students who were registered in the class were eligible to participate and receive this extra

(1) **Background questions:**

- (a) On a scale from 1 to 10, how do you estimate your programming experience?
- (b) On a scale from 1 to 10, how do you estimate your programming experience compared to your classmates?
- (c) On a scale from 1 to 5, how experienced are you with the following programming paradigm: object-oriented?
- (d) Rounded to the closest integer, for how many years have you been programming?
- (e) What kind of computing device did you use for this study?
- (f) Do you have experience in professional software development?

(2) **Code Review Experience Questions:**

- (a) Do you have experience with code reviews in a professional environment?
- (b) Do you have professional experience providing reviews on others' code through GitHub?
- (c) Do you have professional experience receiving feedback from others on your code through GitHub?

(3) **Comparing Code Questions:**

- (a) For how many tasks, approximately, did you consult the following resources: *[none, a few, about half, most, all, I don't know]*
 - (i) Google Search
 - (ii) Q&A forums (e.g., StackOverflow)
 - (iii) Generative AI (e.g. ChatGPT)
 - (iv) Official documentation for APIs
 - (v) IDEs (or similar)
- (b) What was difficult about performing the code review in this study?
- (c) What was easy about performing the code review in this study?
- (d) What would have helped you perform the code review in this study more effectively?

(4) **Demographic Questions:**

- (a) What type of student are you?
- (b) What is your major?
- (c) What is/are your minor(s), if any?
- (d) What is your gender identity? (select all that apply)

Figure 2: Post-activity survey for students; 1a–1d are from a survey instrument in prior work [16].

credit if they completed the pre-survey form (informed consent), the assigned tasks, and the survey instrument (a Google Form). Students could opt-out of having their data used in the study and still receive the extra credit. Students were encouraged to do their best, and we saw no conclusive evidence of students performing this study in bad faith through the checking of submission times and student behavior during the study execution.

Immediately prior to the study, we created a GitHub repository for each student in the class. On the day of the study, students entered the large lecture hall at their regular class time. At the beginning of the study, we gave a five-minute introduction that included a description of code review and a demonstration of how to navigate the GitHub PR interface. Within this introduction, the idea of behavior-perseveration was introduced, and it was reinforced through the README (see Section 3.1.3). Students had the remainder of class time (approximately 70 minutes) to complete the eight tasks and the post-task survey (described in Section 3.4). Students were told to work independently and were allowed to use outside resources such as IDEs, API references, and AI-based tools like ChatGPT or Copilot. All students who consented completed the study tasks and survey within the allotted time.

3.4 Post-study Survey Design

The post-activity survey questions are shown in Figure 2. Section 1 is from an instrument with validated interpretation, which is used

Table 3: Population Demographic Information

Participants	Count	125
Type of Student	Full-time	123
	Part-time	2
Major	Computer Science	113
	Computer Science + Another Major	12
Minor (free-response)	None	76
	Other	35
	Mathematics	14
	Business	12
Gender-Identity (multi-select)	Male	93
	Female	27
	Non-binary	2
	Prefer not to disclose	6

for measuring programming experience in Java, and showed that self-estimation is, on average, a reliable measure of experience [16]. Programming experience refers to any experience programming within schoolwork, internships, or jobs. The questions regarding professional experience with code review in Section 2 were informed by prior work that looked at student experiences with code review [26]. Professional experience was defined as work experience through jobs or internships. Section 3 seeks to understand participant behavior during the study, such as the use of AI-based tools. We asked these questions to evaluate any downstream impacts of those behaviors on accuracy. Section 4 contains demographic questions.

3.5 Participants

Table 3 shows demographic information for the participants. All participants were majors in computer science enrolled in a third-year undergraduate level software engineering course at North Carolina State University. Over the course of a semester, students received instruction in software engineering practices and software engineering tools like GitHub. They also completed two group projects that required the use of GitHub for submitting and reviewing pull requests every week for most of the semester. Through this course and previous courses, students regularly created unit, system, and integration tests as part of their assignments and curriculum.

3.6 Data

We had 125 participants, and all participants completed all tasks, yielding 1,000 initial data points. However, due to an error during the setup of Task D, we had to discard all responses to Task D. This study yielded 875 usable data points.

About half (61/125, 49%) of the participants completely followed the instructions. However, even though we provided a demonstration and reinforced instructions in a README, not all participants followed the instructions for all tasks. Rather than discard these non-standard responses, we created a manual workflow to determine if a response concluded that the program pairs were equivalent or non-equivalent, based on a preponderance of evidence.

The Equivalent Case. Students performed two actions to approve a code change as being behavior-preserving: 1) approve the changes; and 2) merge the change (using the *Merge Pull Request* button in the GitHub PR user interface). In all, 311 tasks "approved" the changes.

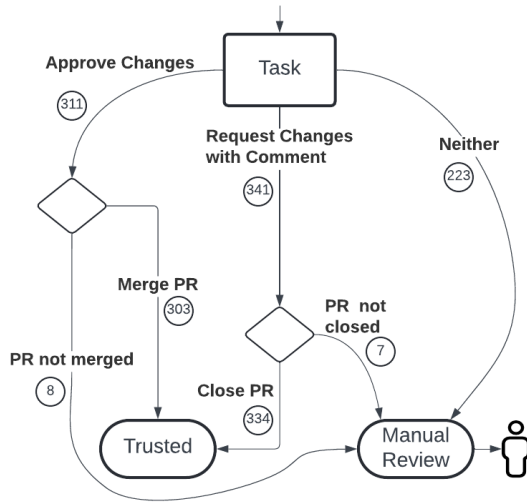


Figure 3: Workflow for handling non-standard responses, as described in Section 3.6.

If participants performed both actions, we mark this particular response *Trusted*, as shown in the workflow diagram in Figure 3; 303 (35%) tasks did both. If only one of those actions was performed (e.g., they approve the PR but do not merge it; 8 tasks (1%), we mark it as *Untrusted* and further inspect the response manually to determine whether the participant’s response clearly indicated equivalent or non-equivalent. Otherwise, we concluded the response was ambiguous, and it was omitted from the final analysis.

The Non-Equivalent Case. Students had to perform two actions to mark a change as behavior-changing: 1) comment on how it changed the behavior as part of the “request changes” response to the PR, and 2) close the pull request. If both actions were performed, we would mark this particular response as being *Trusted*; 334 (38%) did both. If only one of those actions was performed (e.g., they commented, or they closed the PR without comment; 7 tasks (1%), we mark it as *Untrusted* and require further manual investigation.

The Neither Case. If the PR was not marked with “request changes” for non-equivalent PRs or “approve changes” for equivalent PRs, the participant’s intention was not clear. These require further manual investigation, leading to 223 tasks being marked as *Untrusted*.

Manual Review. For manual investigation into the *Untrusted* tasks (238/875, 27%), we did the following: If a task seemed to follow most of the instructions or was just missing one step, then it was marked as *Probably trustworthy*. In addition, if the comments on the PR allowed us to conclude the outcome without any ambiguity, it was marked as *Probably trustworthy*. If we were unable to determine what the outcome was decided on because of a lack of comments or because of a mix of the process (e.g. Accepting changes and then closing the PR instead of merging the PR as instructed), then the task was given a *Probably untrustworthy*. In total, only 22 (3%) tasks were deemed probably untrustworthy and omitted from the analysis. This left 853 (98%) final data points for the analysis.

4 Results

4.1 RQ1: Accuracy of Code Review

To answer this research question, we used the 853 trustworthy responses from the study, as well as the survey responses.

4.1.1 Analysis of Tasks. The overall accuracy was 89%, as shown in Table 1. As the response variable is binary, the null hypothesis compares average accuracy to random, and therefore $H_0 : \mu \leq 0.5$. Using a one-sided t-test, we find that the null hypothesis is rejected with $p < 0.001$. That is, participants did much better than random. To better understand the factors that impact accuracy, we analyze the data using a binomial linear model with contrasts using the predictor responses. We looked for differences across the predictor variables of task, size, depth, order, block, and ground truth.

Impact of Experimental Structure. In the linear model (Section 3.2.1), task order was not significant, and therefore it did not affect accuracy. This means that learning effects were avoided through the block design (Section 3.1.2). The blocks were analyzed for the dependent variable of accuracy. Blocks 13 and 15 had significantly lower accuracy than the other blocks. However, this difference makes sense because both blocks contain tasks with the lowest accuracy in our study, the equivalent versions of Tasks B, F, and H.

Impact of Structural Characteristics of Code. Across program depth and size, using the specified linear model with contrasts, there was no significant difference in accuracy. However, what seemed to matter was the task complexity, as in the *Comprehension* barrier [26]. The only task that had a significant difference in overall accuracy, based on contrasts in the linear model, was Task B. This is likely due to Task B containing recursion, whereas the other tasks had code with simpler structure.

Impact of Ground Truth. We tested whether the ground truth of a task (whether it was equivalent or non-equivalent) introduces significant variation in accuracy. Both the ground truths of non-equivalent and equivalent had significantly different accuracy, using the Mann-Whitney Test ($p < 0.001$). Looking at the means of equivalent (μ_{Eq}) and non-equivalent (μ_{Neq}), our null hypothesis is $H_0 : \mu_{Eq} = \mu_{Neq}$. If $\mu_{Eq} > \mu_{Neq}$, then students overestimate similarity in the program pair, and got more non-equivalent tasks wrong, thinking that they were equivalent. However, if $\mu_{Neq} > \mu_{Eq}$ that means they overestimate differences, the students got more equivalent tasks wrong by thinking there were differences where there were not. In the data, the average accuracy of the equivalent and non-equivalent are $\mu_{Neq} = 93\%$, $\mu_{Eq} = 84\%$, and this difference is significant with $p < 0.001$ per Table 1, meaning that students got more equivalent tasks wrong, overestimating differences.

Impact of Manual Review. As a quality control step, we also looked at the impact of our manual review process (Figure 3). For study responses categorized as *trustworthy* after manual review (219, 84% accuracy) or *untrusted* even after manual review (22, 50% accuracy), there was significantly lower accuracy ($p < 0.05$) than *trusted* responses (637, 91% accuracy). Responses requiring manual review generally did not follow the study processes, potentially introducing a great amount of variance for accuracy. In our study, participants who followed the instructions were the most accurate.

Table 4: Frequency of Generative AI Usage

Category	None	A Few	About Half	Most	All
Responses	51 (40%)	47 (38%)	12 (10%)	12 (10%)	3 (1%)

This suggests our accuracy results may be artificially low in that actual accuracy likely mirrors participants who followed all instructions. Alternately, we hypothesize that stronger students were more likely to follow directions, leading to the set of trustworthy after manual review having a higher accuracy than the untrusted responses after manual review.

4.1.2 Analysis of Participants. Using the participant free-response answers to Questions 3b and 3c in Figure 2, with each row containing a participant, their average accuracy, and their characteristics, we analyzed what factors influenced the accuracy of the code review tasks given on a per-participant basis with average accuracy as the dependent variable.

Impact of Programming Experience. We asked participants about their programming experience using questions 1a–c in Figure 2. We found no evidence that programming experience impacts average accuracy using a linear model with contrasts.

We further examined if participants' accuracy on individual tasks (as opposed to participants' overall accuracy) correlates with reported experience, and did not find evidence relating experience to accuracy. We were somewhat surprised that this was the case even for the lowest-accuracy problem (Task B, limit1). Because programming experience did not impact accuracy on the most difficult problem, we suspect that tasks in this study may have been excessively easy. If tasks were trivial, students (regardless of experience) would be able to perform well, and therefore, programming experience would not matter when it comes to task accuracy. If tasks were non-trivial, we would expect a positive correlation between programming experience and task performance. Thus, the absence of a relationship between experience and accuracy leads us to suspect the tasks could have been too easy, possibly an oversimplification of the tasks.

Impact of Generative AI. Many students used some kind of generative AI within the eight code review tasks and reported it through the post-study questionnaire, shown in Table 4. A majority of students (74/125, 59%) used generative AI on at least a few of their code review tasks, but 51 (41%) reported they did not use generative AI at all. Looking at accuracy, we find that the 15 participants who used generative AI for *most* or *all* of the tasks had significantly lower accuracy 79% ($p < 0.05$, Mann-Whitney) when compared to everyone else, though the absolute number of participants is small.

RQ1 Summary: Students were highly successful (89%) in recognizing changes in code behavior. Compared to prior work, this higher accuracy could be the result of simplification.

4.2 RQ2: Student mistakes when identifying changes in behavior

Students made a variety of mistakes. When a student marked a task as non-equivalent when it was truly equivalent, we analyzed the

Table 5: Student's Foundational Mistakes in Code Review

Categories	A	B	C	E	F	G	H	Total
Incomplete evaluation of change logic	3	13	1	0	1	0	1	19 (42%)
Misunderstanding of Behavior Preservation	0	1	1	1	8	0	7	18 (40%)
Unclear	1	4	0	0	0	1	2	8 (18%)

comments. There were 45 (5%) such tasks; out of those, two major groups of mistakes arose. Table 5 shows the mistakes and how they were distributed among the tasks.

4.2.1 Shallow/Incomplete evaluation of change logic. These students often spent time unraveling the logic or structure underlying the code change but were unable to identify correctly that the change was equivalent. For example, in Task A, (Figure 4) the change is that the new conditional structure is equivalent to the previous one. However, some students were unable to grasp that change in structure, saying things like “if absb != 0.0, the output will be different” However, this student fails to recognize that no matter the value of absb, the value returned will be the same, but with a different control flow.

4.2.2 Misunderstanding of Behavior Preservation. Some students misunderstood the concept of behavior preservation within a code change and how it applies within a code review setting. These students often looked at surface-level changes and called those non-behavior-preserving changes or judged the changes by a different heuristic, like the number of computations done. For example, in the equivalent version of Task C (shown in Figure 5), one student says, “This changes the code as now more computations are done, and different computations are done. This includes computing -y which was previously not done.” This suggests that there was a misunderstanding of what causes behavior to be changed or preserved within the code snippet, perhaps adopting an algorithmic-complexity approach to determining equivalence rather than a behavioral one.

4.2.3 Other Mistakes. There were three participants who made simple *logic mistakes* and deemed the programs non-equivalent based on making invalid test cases or misreading the code. The last eight participants did not leave enough information to understand their thought process, so it was *unclear* what went wrong.

RQ2 Summary: After simplifying barriers, student mistakes fell into two main categories: 1) misunderstandings about the nature and definition of behavior preservation; and 2) reasoning about the consequences of code changes.

4.3 RQ3: Simplifying Barriers to Comparative Comprehension

Based on the free response answers from question 3b in Figure 2, “What was difficult about performing the code review in this study?” we performed a closed card sort, mapping difficulties to the barriers in Section 3.1.4.


```

1 public static double snippet (double a, double b) {
2     double absa = 0;
3     double absb = 0;
4     absa=Math.abs(a);
5     absb=Math.abs(b);
6     if (absa > absb) {
7         return absa*Math.sqrt(1.0+SQR(absb/absa));
8     }
9     else {
10         if (absb == 0.0 )
11             return 0.0;
12         else
13             if (absb != 0.0 )
14                 return absb * Math.sqrt(1.0 + SQR(absa / absb));
15             else
16                 return absb;
17     }
18 }
19 public static double SQR(double a) {
20     return a*a;
21 }

```

Figure 4: Non-equivalent Programs used in Task A

```

11 - if (y > 8) {
12 + if (-y < -8) {

```

Figure 5: Code Change for Equivalent Programs in Task C

For the closed card sort for question 3b, there were two coders with 125 cards. There was an initial agreement on 78 cards, and disagreement on 47 cards, with a Cohen’s Kappa inter-rater reliability rating [25] of 0.62, representing a *moderate-to-substantial* level of agreement. After the initial pass, there was a discussion of all the cards, and based on this discussion, we created a new comparative comprehension category, *Context Comprehension*, and then re-applied it with both coders working together. In the end, both coders came to a consensus. The results are shown in Table 6. As shown, all previous barriers have lower frequency in our study. This lower frequency could be due to the study context or due to the simplification of the barriers.

Table 6: Results of Closed Card Sort on Question “What is difficult about performing this code review in this study?”

Barrier	Count	%	Prior Frequency
Comprehension	21	17%	48%
Delta Comprehension	20	16%	34%
Large Scope	14	11%	25%
Unfamiliar Code	6	5%	48%
Limited/Misaligned View	5	4%	39%
Self-Doubt	5	4%	7%
Unclear Motivation	4	3%	7%
Limited Time	3	2%	9%
Toolchain Issues	3	2%	1%
Deep Changes	1	1%	16%
Context Comprehension	28	22%	N/A
N/A	12	10%	N/A
Other	10	8%	N/A

From the closed card sort of question 3b, we found that the barriers from the previous work needed a slight revision. Many answers mentioned not knowing the context of the code, mentioning that even though they had the entire program, they didn’t know the environment surrounding the file. In the prior work, participants were familiar with the codebase, so this barrier was not present by design. However, for us, it was rather common. We call this new barrier *Context Comprehension* and it falls within the category of the *Code Comprehension* barriers. We define it as follows:

Context Comprehension: *Understanding the context surrounding a code change, including the context of the original code in terms of how it is used and why it is important.*

Essentially, if the user does not know how the program is used, what are the common use cases, or how it interacts with other parts of the system, we found they struggle to understand the impact of the change being made. Further, this blocks them from discerning comfortably if that change was behavior-preserving or not.

From Table 6, the frequency of *Comprehension*, *Deep Changes*, *Delta Comprehension*, *Limited/Misaligned View*, and *Unfamiliar Code* decreased, suggesting that we were able to successfully reduce those barriers. One notable barrier that did not appear in Table 6 was *Lack of Tests*. Previous studies show the importance of tests within the code review and pull request process [45] [44] [17]. Yet for this study, students performed well despite the absence of tests.

RQ3 Summary: Participants reported fewer barriers to comparative comprehension in our study, demonstrating that it is possible to simplify barriers and facilitate students’ success in the code review of behavior-preserving changes.

5 Discussion

We discuss the study results in the context of prior work, implications for SE educators, and the impact of generative AI on tasks.

5.1 Over-estimating Differences

We found in RQ1 that students got more equivalent tasks wrong, hence they were perceiving differences where none existed. This finding comes as a contrast to previous work [28] where students overestimated *similarities*. Here, we explore a potential explanation.

It is possible participants in our study misunderstood what behavior-preserving changes are, but upon further investigation, that number appears small (Table 5). It is possible that in simplifying some common barriers such as *Comprehension* and *Unfamiliar Code*, which are about not being able to understand code within a code review (Table 6), the tasks became too simple, leading students to second-guess their intuition and create differences where none existed. It is possible that by simplifying barriers such as *Large Code* and *Limited/Misaligned Views* that deal with challenges in reviewing large amounts of code, students were less likely to throw in the proverbial towel and think, “looks close enough to me.” Instead, they thought critically about the differences and identified the behavioral changes accurately. In fact, research on code review would suggest this is the case, as smaller changes are easier to review [37]. In the end, we seem to have succeeded in creating a context for

comparative comprehension that sets students up for success by simplifying barriers.

5.2 Suggestions for SE Educators

Code review is a valuable learning tool that can enable students to learn actively, justify their work, and give feedback [20]. Based on our results, we suggest the following ideas and processes for SE educators in code review.

5.2.1 Simplified Tasks. We found that students can understand the core of comparative code comprehension within the activity of code review with fewer impediments when a simplification process is used. For example, in a CS1 class, code review activities similar to the tasks posed in this study could be assigned, allowing students to practice their core competencies of individual source code comprehension, comparing the delta, and detecting if the change is behavior-preserving. While this may provide tasks that are artificially too easy, as seen in 4.1.2 and may leave crucial elements out due to the nature of simplification [46], this will allow students to focus on the fundamental skill of comparative code comprehension.

5.2.2 Use of GitHub and other tools within the classroom. Through our analysis of survey question 3b (as seen in Table 6), the use of GitHub and the “diff” view provided an easy-to-understand interface for students to view the difference between two pieces of code, decreasing the number of people who mentioned the barrier “Limited or Misaligned View” (Section 3.1.4) by 35%, the second highest difference when compared to the prior study. We recommend that GitHub and other tools be used within the classroom as they teach modern skills to computer science students and can help alleviate issues surrounding view or user interfaces relating to code review.

5.2.3 Importance of Context within the Process of Comprehension. The creation of the new barrier “Context Comprehension” highlights the important role that context plays in code review. Therefore, we recommend clearly communicating the context of the code repository during code review tasks. This can look like having the students do a code review of PRs from a known repository or giving the students assignments to get to learn about the repository prior to code review. The objective of giving this context is to help students understand how the system is used and how each file interacts with each other, causing them to understand the impact of the changes made more fully.

5.3 Impact of Generative AI

In the analysis of the study, for those who used generative AI the most, their accuracy was lower than the rest of the population (Section 4.1.2). It is possible that those who used generative AI the most engaged the least with the study tasks.

As no measure of AI usage was recorded other than students’ self-reported responses, however, we do not have a way to triangulate this finding. At the same time, we have no reason to believe that students were untruthful in their responses for the following reason: during the class in which the study took place, students were allowed to use generative AI as long as they declared it. Even so, this measure is subject to recall bias as students performed the tasks and then had to recall to what extent they used generative AI within the post-study.

6 Threats to Validity

Internal Validity. We conducted this study in two separate sections of the third-year undergraduate software engineering class, one in the morning (8:30 a.m.) and one in the afternoon (3:00 p.m.). The accuracy in the morning section was 90% while the afternoon was 85%, but the difference was not significant ($p = 0.07$). The commit message explaining the PR, “Update code to adhere to code standards,” is provided to help motivate the changes, but the actual changes do not appear related to any actual code standards, so participants might still experience the *Unclear Motivation* barrier. The use of generative AI introduces a factor influencing the student’s cognitive process. The accuracy numbers could be a result of generative AI, the cognitive process of the student, or a combination of both. However, due to the lack of correlation found between the amount of generative AI used and the accuracy on code review of behavior-preserving changes, the cognitive process of the students may still be modeled correctly.

External validity. We used only eight programs to simplify the code review task, and this simplification may have trivialized the task in certain aspects, a common threat in the simplification methodology [46]. Additionally, our study only showed programs in Java, which may not generalize to other languages. The changes between programs that we presented to students are small; therefore, these tasks may not generalize to real-world code changes. Students were allowed to use whatever devices they had with them to participate in this study. A modality like a tablet or mobile device could pose a threat to validity due to the limited interface and interactions possible with the device. However, most participants (120/125, 96%) used just laptops to complete this study with very few using other modalities like mobile devices, or tablets. Only 3 (2%) participants used tablets or mobile devices and resulted in a cumulative average accuracy of 95%, higher than the average accuracy for the entire dataset.

7 Conclusion

With an overall task accuracy of 89%, this work successfully simplified barriers, and students succeeded in identifying behavior-preserving changes in code review. Based on the mistakes students made, we conclude that the two most foundational errors in this task are: 1) misunderstanding the concept of behavior preservation; and 2) incorrect reasoning about the consequences of code changes. These results have implications for tool support and SE education aimed at code change comprehension, especially in the context of code review. Simplifying these barriers allows students to see and comprehend the code unhindered, creating a space where students can focus on understanding the changes. Since comparative comprehension is vital to modern code review, simplifying barriers must be done to help students learn the process.

Acknowledgements

This work is funded in part by NSF SHF #2141923, #2006947, and #1749936. Special thanks to Dr. Emily Griffith for her statistical expertise and the students of North Carolina State University’s Software Engineering course for their participation in our study.

References

- [1] Seham Alharbi and Dimitris Kolovos. 2024. Exploring the Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension* (Lisbon, Portugal) (ICPC '24). Association for Computing Machinery, New York, NY, USA, 236–240. <https://doi.org/10.1145/3643916.3644395>
- [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. <https://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. <https://doi.org/10.1145/3524842.3527932>
- [4] Barış Ardic, İrem Yurdakul, and Eray Tüzün. 2020. Creation of a Serious Game for Teaching Code Review: An Experience Report. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEET)*. 1–5. <https://doi.org/10.1109/CSEET49119.2020.9206173>
- [5] 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.
- [6] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 610–614. <https://doi.org/10.1109/MSR52588.2021.00084>
- [7] 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.
- [8] Krzysztof Borowski, Bartosz Balis, and Tomasz Orzechowski. 2024. Semantic Code Graph—an information model to facilitate software comprehension. *IEEE Access* (2024).
- [9] Chun Yong Chong, Patanamom 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.
- [10] Albert Danial. 2024. cloc. <https://github.com/AlDanial/cloc>
- [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] Angela Dean and Daniel Voss. 1999. *Design and analysis of experiments*. Springer.
- [13] Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. 2007. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 281–285.
- [14] Asaf Etgar, Ram Friedman, Shaked Haiman, Dana Perez, and Dror G. Feitelson. 2022. The effect of information content and length on name recollection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (Virtual Event) (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 141–151. <https://doi.org/10.1145/3524610.3529159>
- [15] Juan Carlos Farah, Basile Spaenlehauer, Vandit Sharma, María Jesús Rodríguez-Triana, Sandy Ingram, and Denis Gillet. 2022. Impersonating Chatbots in a Code Review Exercise to Teach Software Engineering Best Practices. In *2022 IEEE Global Engineering Education Conference (EDUCON)*. 1634–1642. <https://doi.org/10.1109/EDUCON52537.2022.9766793>
- [16] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 73–82. <https://doi.org/10.1109/ICPC.2012.6240511>
- [17] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering (Hyderabad, India) (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 345–355. <https://doi.org/10.1145/2568225.2568260>
- [18] 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 (Gothenburg, Sweden) (ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 1–4. <https://doi.org/10.1145/3183377.3183393>
- [19] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating pedagogical code reviews into a CS 1 course: an empirical study. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) (SIGCSE '09). Association for Computing Machinery, New York, NY, USA, 291–295. <https://doi.org/10.1145/1508865.1508972>
- [20] 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.
- [21] Philipp Kather and Jan Vahrenhold. 2021. Is Algorithm Comprehension Different from Program Comprehension?. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 455–466. <https://doi.org/10.1109/ICPC52881.2021.00053>
- [22] 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.
- [23] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwinka. 2017. Code reviewing in the trenches: Challenges and best practices. *IEEE Software* 35, 4 (2017), 34–42.
- [24] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. <https://doi.org/10.1214/aoms/1177730491>
- [25] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
- [26] Justin Middleton, John-Paul Ore, and Kathryn T Stolee. 2024. Barriers for Students During Code Change Comprehension. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 196, 13 pages. <https://doi.org/10.1145/3597503.3639227>
- [27] Justin Middleton, Neha Patil, and Kathryn T Stolee. 2024. Co-Designing Web Interfaces for Code Comparison. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 187–198.
- [28] 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.
- [29] Mika V. Mäntylä and Casper Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 3 (2009), 430–448. <https://doi.org/10.1109/TSE.2008.71>
- [30] Delano Oliveira, Reynold Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-Centric Studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 348–359. <https://doi.org/10.1109/ICSME46990.2020.00041>
- [31] Eduardo Oliveira, Shannon Rios, and Zhuoxuan Jiang. 2023. AI-powered peer review process: An approach to enhance computer science students' engagement with code review in industry-based subjects. In *2023: ASCILITE 2023 Conference Proceedings: People, Partnerships and Pedagogies*. <https://doi.org/10.14742/apubs.2023.482>
- [32] 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.
- [33] 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.
- [34] 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.
- [35] Carol Righi, Janice James, Michael Beasley, Donald L Day, Jean E Fox, Jennifer Gieber, Chris Howe, and Laconya Ruby. 2013. Card sort analysis best practices. *Journal of Usability Studies* 8, 3 (2013), 69–89.
- [36] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.
- [37] 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 (Gothenburg, Sweden) (ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 181–190. <https://doi.org/10.1145/3183519.3183525>
- [38] 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.
- [39] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, Fabio Palomba, and Filomena Ferrucci. 2022. Toward Understanding the Impact of Refactoring on Program Comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 731–742. <https://doi.org/10.1109/SANER53432.2022.00090>
- [40] Daniel Augusto Nunes da Silva, Daricélio Moreira Soares, and Silvana Andrade Gonçalves. 2020. Measuring Unique Changes: How do Distinct Changes Affect the Size and Lifetime of Pull Requests?. In *Proceedings of the 14th Brazilian*

- Symposium on Software Components, Architectures, and Reuse* (Natal, Brazil) (SB-CARS '20). Association for Computing Machinery, New York, NY, USA, 121–130. <https://doi.org/10.1145/3425269.3425280>
- [41] 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* (Trondheim, Norway) (ITiCSE '20). Association for Computing Machinery, New York, NY, USA, 173–179. <https://doi.org/Song2020Peer>
 - [42] Saikrishna Sripada, Y. Raghu Reddy, and Ashish Sureka. 2015. In Support of Peer Code Review and Inspection in an Undergraduate Software Engineering Course. In *2015 IEEE 28th Conference on Software Engineering Education and Training*. 3–6. <https://doi.org/10.1109/CSEET.2015.8>
 - [43] 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* (Cary, North Carolina) (FSE '12). Association for Computing Machinery, New York, NY, USA, Article 51, 11 pages. <https://doi.org/10.1145/2393596.2393656>
 - [44] Erik van der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) (MSR '15). IEEE Press, 357–361.
 - [45] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) (ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 805–816. <https://doi.org/10.1145/2786805.2786850>
 - [46] Henry G Widdowson. 1978. The significance of simplification. *Studies in Second Language Acquisition* 1, 1 (1978), 11–20.
 - [47] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 years of designing code comprehension experiments: A systematic mapping study. *Comput. Surveys* 56, 4 (2023), 1–42.
 - [48] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4. <https://doi.org/10.1109/ICPC.2016.7503738>