Code Review Comprehension: Reviewing Strategies Seen Through Code Comprehension Theories

Pavlína Wurzel Gonçalves
Department of Informatics
University of Zurich
Zurich, Switzerland
p.goncalves@ifi.uzh.ch

Pooja Rani
Department of Informatics
University of Zurich
Zurich, Switzerland
rani@ifi.uzh.ch

Margaret-Anne Storey
Department of Computer Science
University of Victoria
Victoria, Canada
mstorey@uvic.ca

Diomidis Spinellis
Department of Management Science and Economy
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

Alberto Bacchelli
Department of Informatics
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Despite the popularity and importance of modern code review, the understanding of the cognitive processes that enable reviewers to analyze code and provide meaningful feedback is lacking. To address this gap, we observed and interviewed ten experienced reviewers while they performed 25 code reviews from their review queue. Since comprehending code changes is essential to perform code review and the primary challenge for reviewers, we focused our analysis on this cognitive process. Using Letovsky's model of code comprehension, we performed a theory-driven thematic analysis to investigate how reviewers apply code comprehension to navigate changes and provide feedback.**

Our findings confirm that code comprehension is fundamental to code review. We extend Letovsky's model to propose the Code Review Comprehension Model and demonstrate that code review, like code comprehension, relies on opportunistic strategies. These strategies typically begin with a context-building phase, followed by code inspection involving code reading, testing, and discussion management. To interpret and evaluate the proposed change, reviewers construct a mental model of the change as an extension of their understanding of the overall software system and contrast mental representations of expected and ideal solutions against the actual implementation. Based on our findings, we discuss how review tools and practices can better support reviewers in employing their strategies and in forming understanding.

Data and material: **https://doi.org/10.5281/zenodo.14748996**

***Index Terms*—Human Factors, Code Review, Code Comprehension, Code Review Strategies**

I. Introduction

Modern code review is a widely used practice to ensure the quality of code contributions [41, 23, 16]. Typically, developers manually review code changes through an informal, tool-based, and asynchronous process [23, 41]. Reviewing code changes is seen as beneficial to identify defects and find alternative solutions, improve transparency in the team, and share knowledge among developers [2]. Despite its benefits, code review is time-consuming [2], challenging to adopt [5], and costly to sustain [35]. Therefore, researchers have been focused on investigating ways to support developers in performing code reviews efficiently and effectively.

Code comprehension [44, 24], particularly comparative code comprehension [34], is the most important competency reviewers need [52] and simultaneously their greatest challenge [2]. However, there is little insight into how developers form their understanding during code review and use it to provide feedback. Understanding these aspects of individual code review performance can inform the design of code review tools and practices [51] that support reviewers' construction of the mental models [47] and in following effective individual reviewing strategies [30]. Human-centric design can improve developers' productivity in code review and their experience when facing common challenges, like code change complexity [29].

In this study, we investigate in detail how developers form and use their understanding of the code change under review. Previous work has studied the behavior of reviewers using various methods, ranging from controlled experiments [22, 21, 18], to eye-tracking [48], and analyses of traces left in software repositories [9, 36, 21]. These studies describe reviewing mostly as linear reading of code [21, 48]. However, the strategy for reviewing large changes remains unexplained (*e.g.*, in terms of reading order [6]). These studies often involve participants reviewing small code changes from unknown systems and lack a real context, purpose, and interactions among authors and reviewers. As an alternative, we selected observation accompanied by think-aloud protocols and interviews to capture the reviewer's decisions or comprehension strategies in real time [1, 49]. We observed ten experienced developers—recommended by others as great reviewers—while performing 25 code reviews in their work environment to gain insight into their reviewing strategies. We performed an in-depth theory-driven qualitative analysis to code and interpret the collected data, using several theories from code comprehension and psychology [32, 38, 40, 3].

As a result of our observations, we contribute with an extension of Letovsky's code comprehension model [32]: a *Code Review Comprehension Model*. Our model describes and

enhances our understanding of how reviews are performed and of the role of code comprehension in shaping the review process. The model highlights the opportunistic nature of code review comprehension strategies employed by reviewers to deal with review complexity, *e.g.*, scoping down the review or employing other code inspection strategies apart from linear reading, such as chunking or segmenting code based on reviewing difficulty. We also identify the knowledge and information sources that help reviewers navigate code, construct a mental model of the pull requests (PRs), and provide feedback by comparing their mental model of the PR to ideal and expected solutions. Finally, based on our observations and analyses, we propose guidelines for performing code review and designing better human-centered review tools.

II. Background

We provide details on theories and findings we used in the qualitative analysis to code and interpret the data.

Opportunistic strategies to code comprehension. Code comprehension research has observed, described, and interpreted code navigation strategies aimed at understanding a code artifact [10, 11, 13, 24]. Accordingly, researchers have proposed numerous comprehension models to capture the comprehension process [10, 44, 32, 49]. For instance, researchers found that developers initially take a top-down comprehension approach to gain an overall big picture and then move to detailed code snippet [10]. The top-down approach relies on applying domain knowledge, knowledge of programming plans, and rules of programming discourse to interpret specific implementations of programmatic solutions. Developers also use other strategies for code navigation, for example, bottom-up [50, 44] or following the control flow [37].

Developers *opportunistically* choose the strategies to combine the top-down and bottom-up processes and seek the information relevant to their task to form understanding most efficiently and update their knowledge through the comprehension process [31, 32]. In addition to understanding code, code comprehension is a fundamental step in supporting subsequent software development activities [50]. Relevant to our study, code comprehension also determines how developers review code, the review outcomes, and the artifacts created in the process. We aim to understand which approaches are used and how the mental model of the reviewed code change is formed.

Role of experience in code review comprehension. Code review is based on forming an accurate mental model of the change and ensuring maintainability by other developers [52]. Developers achieve understanding through recognition of programming plans, *i.e.*, stereotypical implementations of goals [44]. Reviewers' experience is crucial to recognize and correct programming patterns into readable and maintainable code: Novice developers tend to have more difficulty understanding a program [7], gaining the overall picture, and identifying programming patterns [27]. Consequently, adherence to coding conventions and programming patterns ensures also adherence to representations shared throughout the developer community and thus, fitting the mental schemas of other developers. In the analysis, we have viewed reviewers as gatekeepers ensuring code understandability and maintainability by promoting adherence to these shared standards.

Code Review Strategies. Past research has provided evidence that reviewers often perform *ad hoc* reviews [12]: They "just read the code" or adopt an unsystematic approach that relies on personal strategies to navigate code and identify defects [12, 48]. Baum et al. [6] found that reviewers mostly navigate the files linearly following the order offered by the review tool. Fregnan et al. [21] found a linear decline in the number of comments reviewers leave as the files appear later in a change set, and measured diminished developers' effectiveness at detecting defects in files presented last by review tools. In eye-tracking experiments, reviewers were found to perform reviews mostly linearly, splitting the review into a scanning phase (where reviewers first get an overview of the code) and a detailed phase (where they return to look into specific code segments) [48].

Although review navigation seems to be often linear, this may not be the case for more complex reviews. In fact, complex changes are more challenging to review [29] and the order in which reviewers choose to review them remains largely unexplained [6]. For example, Spadini et al. [45] indicate that reviewers might choose to follow diversified review strategies, such as performing test-driven reviews. Therefore, in our study we observe reviews of changes of varying complexity to better understand how complexity may affect the process.

Code comprehension and code review. Similarly to code comprehension, it seems reasonable to expect that reviewers performing ad hoc reviews [12] may shape their reviewing process through a combination of strategies and opportunistic decisions that aim to optimize for both effective understanding and efficient reviewing. However, code review requires deeper engagement of higher-level cognitive processes (*e.g.*, decision making and analysis) than code comprehension alone due to the need to inspect the code changes for quality aspects [20].

Code review is a specific context for using code comprehension, as understanding occurs within an environment where the changes to the software system are *iterative* (reviewers repeatedly comprehend the same artifact), *incremental* (they comprehend a modification of a likely partially known software system), and *interactive* (the comprehension is supported by interactions with the author and other colleagues). Reviewing may require developers to work with other software artifacts, such as the PR discussion, issues, testing tools, architecture, and design documentation. Therefore, comprehension models that address only code navigation may be insufficient to fully capture code review comprehension.

III. Methodology

Our study investigates how software developers perform code reviews through the lens of code comprehension theories. As the main tool for approaching the data and informing the research questions, we use Letovsky's model of code

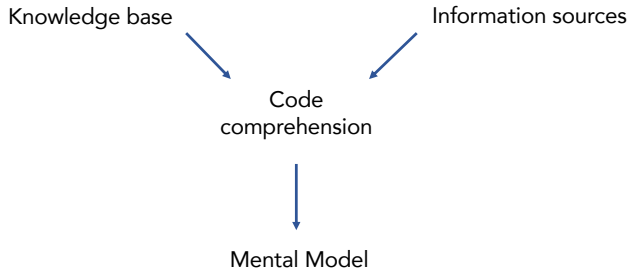


Fig. 1. Letovsky's Model of Code Comprehension [32] - Code comprehension is an assimilation process using knowledge base and information sources to create a mental model of the code.

comprehension [32] (shown in Figure 1). The model is based on Piaget's model of cognitive development [38], which posits that learning is an activity with a *scope*, as the ability to learn in a certain moment has a limit of information that can be effectively processed.

A. Research Questions

We address three research questions. First, we ask:

RQ$_1$. *How do reviewers scope code review comprehension?*

Although reviews are often seen as unsystematic [12, 48], their reliance on code comprehension [2] suggests that reviewing may follow opportunistic strategies as code comprehension does. Understanding review strategies and the decisions in their selection, particularly when handling complex changes [6], can provide valuable insights on how to support reviewers to perform effective code reviews and inform the design of more human-centered review tools. In RQ$_2$, we focus on the Code Comprehension component of Letovsky's model:

RQ$_2$. *What strategies do developers use to perform code review?*

In Letovsky's model of code comprehension [32] (see Figure 1), code comprehension is accompanied by other three components of the model: (1) the information sources including the code artifact being understood and (2) the knowledge base, which interact in the code comprehension process to create (3) the mental model of the code artifact. To capture how code comprehension is used in code reviews, we also ask:

RQ$_3$. *What are the roles of information sources, knowledge base, and mental models in the code review process?*

B. Data Collection

Given that the cognitive processes underlying a code review are not directly observable, we use observation, interviewing, and think-aloud protocols to capture the reviewer's cognitive process similarly to other studies looking into code comprehension or developers' approaches to perform engineering tasks [1, 49]. Throughout the paper, participants and specific review sessions are referenced using the format [participant ID][review number] (*e.g.*, P2R3).

a) Recruitment of Participants: Following estimates for grounded theory studies, we expected to need 20 to 30 observations to reach saturation [14]. We used purposeful sampling method [4] to achieve high variety in the data by observing 1) reviews in a real-world context, thus observing more varied PRs, 2) reviews of varying complexity and 3) reviews in both OSS community and in in-house software development teams.

We recruited reviewers recommended as great reviewers or who had at least ten years of code review experience. This approach allowed us to observe experienced reviewers with purposeful decision-making in code reviews who have high knowledge of programming patterns and are recognized by other people as insightful. We specifically obtained six reviewers through their companies—four from an open-source software development company (participants P1, P3, P4, P5) and two from a closed-source development company (P8 and P10). We asked developers within these companies to recommend great reviewers, and researchers only received contact details for those who had been recommended and agreed to participate. Additionally, four reviewers were contacted individually: P2, an experienced author of open-source software (OSS) and of technology books and academic publications on software quality; P6, an experienced OSS reviewer recommended by P2; P7, a software engineer at a big technological firm who shares their coding and reviewing practices on their YouTube channel; and P9, an insightful and experienced reviewer known from a previous study. The reviewers typically held leadership and mentoring roles in their teams, had over ten years of professional experience, and reviewed code daily (see Table I).

b) Review&Interview sessions: We collected data in three phases – (1) observation of code review sessions, (2) exploratory interview, and (3) a demographic survey. We conducted each session through an online video call that lasted approximately one hour. With the consent of the participants, we recorded each session for subsequent data analysis. During each session reviewers performed both a short and a long review on PRs from their review queue (at work or in their open source project). While reviewing, participants used a think-aloud method. The interviewer played a passive role, mainly intervening only to facilitate the think-aloud. After each review, the researcher conducted a short semi-structured interview to clarify the observations – establishing the sequence of actions/steps in the review, reasoning and choices of the reviewers and differences among participants. These areas were captured in a non-binding interview guide. Following these sessions, the participants completed a short demographic survey.

As a result of our data collection process, we acquired rich data from 25 review sessions. Table II details the reviewed changes: Their sizes ranged from 2 to 34,520 changed lines of code, and the reviews were conducted using GitHub (N=18), GitLab (N=2), Phabricator(N=2), or Gerrit(N=3). We observed 15 reviews in open-source and 10 in closed-source contexts. In most reviews, our participants ended their task by requesting changes (N=11) or providing comments (N=9), while they directly approved a minority (N=5).

c) Transcription: We transcribed the observations and interviews to enable the subsequent analyses. For the observations, we created the transcripts based on what was vocalized by the reviewers in their think-aloud and by actions performed in the reviewing platform. As a validation of this transcription process, the second author of the study compared three transcripts (P2R2, P4R1, P9R1) against the non-anonymized observations, providing feedback to the first author and pointing out additional insights. These were used to improve and update the other transcripts.

C. Data Analysis

Through familiarizing with the collected observations and interviews, we confirmed that themes of understanding and comprehension were prominent and that code comprehension theory, particularly Letovsky's model of code comprehension [32], provides us with an effective vocabulary to capture the scope of our observations.

Therefore, we used Letovsky's model complemented by other models, terms and theories from code comprehension and psychology [38, 3] to produce the coding schema presented in Figure 2, through which we performed a theory-driven thematic analysis [8].

- *Letovsky's code comprehension model* [32] (Figure 1): captures code comprehension as an interplay of information sources and knowledge base through which the developer creates a mental model representing the code artifact. Mental models consist of the specification, annotation and implementation layer.
- *Piaget's model of cognitive development [38]:* posits that learning is an activity with a *scope* that happens in achievable increments rather than being formed as a complete and comprehensive understanding of a new input.
- *Self-discrepancy theory [3]:* interprets human distress as a result of a discrepancy between their self perception and their own and societal ideals and expectations. We applied it to interpret how reviewers use discrepancies between the PR and their expectations and ideals to interpret and evaluate the code change.

The second author validated the text excerpts using transcripts from three review sessions (P2R2, P4R1, P9R1). Within the concept-related excerpts, we proceeded with a bottom-up coding [8] to understand how these concepts can be described and understood in the code review context. Subsequently, we used the research work mentioned in Section II as a basis to interpret the data.

The analysis aimed to reach code saturation [25]. Therefore, the themes reported in the paper represent a coding structure that was stable and repetitive by the end of the analysis.

We performed the analysis using the qualitative research software NVivo 14. The first author was the person collecting and analyzing the data. To support the validity of the findings, we used *peer debriefing* [15]. Theories and preliminary results were regularly discussed with the second author who also performed checks for validity of transcripts and first-level coding. We discussed results and interpretations among all the authors at two major milestones: (1) during the analysis and (2) once the findings were interpreted and reported.

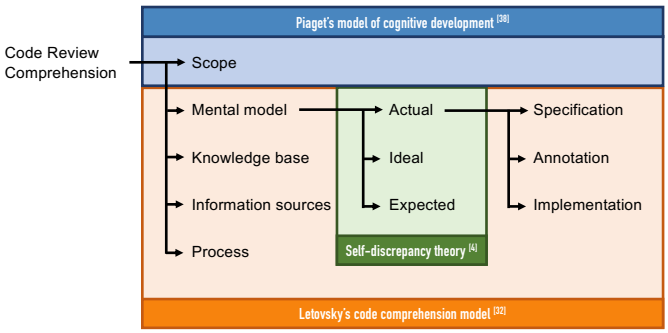


Fig. 2. Coding schema used to structure and code the data according to the selected code comprehension and psychological theories.

Our replication package includes the observation and interview transcripts, interview guide, coding schema, and other documents [53].

D. Ethics and Data Handling

The Human Subjects Committee of the home university of the first, second, and last author approved the study design. Reviewers signed an informed consent before participating in a monitored review session. Observations including participants' screens and faces were temporarily stored on the university server to create anonymized behavioral transcripts. The observed code came from OSS projects or was shared with the permission of the project/team lead, given no sensitive information was shared in the recorded code.

E. Limitations

The scope of the study focused on describing aspects related to code comprehension, using observation of reviews performed by experienced reviewers performed in a real-world context. There are limitations to this focus.

Being the first observational study of cognitive processes in code review, our data collection was initially exploratory and focused on aspects of code comprehension only in the analysis phase. We did not delve deeply into other key aspects of code review, such as reviewers' interactions—a possible rich field for further exploration.

We conducted the observations via a video call. We asked participants whether their review process was the same as usual. Only P2 mentioned they were highly aware of being observed and possibly made their review more thorough than usual, suggesting a possible influence of the Hawthorne effect [17]. To capture natural strategies, we avoided interrupting the reviewers, keeping the clarifications for the end of the sessions. This approach allowed us to observe the higher level flow, rather than to investigate details of each decision made, *e.g.*, reasoning for writing individual comments.

The study participants are experienced reviewers with established reviewing strategies and reasoning. Therefore, this data does not represent the strategies of novice reviewers.

TABLE I
DESCRIPTIVE DEMOGRAPHICS OF THE STUDY PARTICIPANTS

Participant ID	Role	Gender	Experience (years) programming	Experience (years) reviewing	Team/Project tenure (years)	Frequency of programming	Frequency of reviewing
P1	Technical Lead	Female	17	17	3	Daily	Daily
P2	Code base author	Male	37	10	1	Weekly	Monthly
P3	Project Lead	Male	17	5	3	Daily	Daily
P4	Team Lead	Male	16	15	10	Daily	Daily
P5	*Chose not to disclose*						
P6	Core contributor	Male	14	16	13	Daily	Daily
P7	*Chose not to disclose*						
P8	Senior developer	Male	3	2	1	Daily	Daily
P9	Team Lead	Male	11	10	4	Daily	Daily
P10	Team Lead	Male	15	9	2	Daily	Daily

TABLE II
DESCRIPTIVES OF THE REAL-WORLD REVIEWS CONDUCTED BY OUR PARTICIPANTS DURING THE OBSERVATIONS.

Review ID	Tool	Code	Iteration(*)	Draft	Verdict	Number of reviewers	Number of files changed	Number of lines added	Number of lines removed
P1R1	Gerrit	CSS	2nd	✖	Request changes	5	3	97	86
P1R2	GitHub	OSS	1st	✖	Comment	3	1	15	21
P1R3	Gerrit	CSS	1st	✖	Request changes	4	6	174	2
P2R1	GitHub	OSS	1st	✖	Request changes	1	13	1,366	76
P2R2	GitHub	OSS	2nd	✖	Accept	1	8	376	45
P2R3	GitHub	OSS	3rd	✖	Request changes	1	6	109	39
P2R4	GitHub	OSS	1st	✖	Accept	1	1	13	4
P3R1	GitHub	OSS	1st	✖	Accept	1	1	4	4
P3R2	GitHub	OSS	1st	✔	Request changes	1	7	369	98
P4R1	GitHub	OSS	1st	✖	Accept	1	1	1	1
P4R2	GitHub	OSS	1st	✔	Comment	1	3	158	92
P5R1	Gerrit	OSS	1st	✖	Request changes	3	12	79	0
P5R2	GitHub	OSS	2nd	✔	Comment	2	36	3,848	40
P6R1	GitLab	OSS	1st	✔	Comment	1	4	49	6
P6R2	GitLab	OSS	1st	✖	Comment	1	6	360	3
P7R1	Phabricator	OSS	1st	✖	Request changes	3	3	3	0
P7R2	Phabricator	OSS	3rd+	✖	Comment	6	5	57	36
P8R1	Github	CSS	3rd+	✖	Accept	2	1	41	0
P8R2	Github	CSS	3rd+	✖	Request changes	3	17	535	2
P9R1	Github	CSS	1st	✖	Comment	1	17	(**)17,030	(**)17,490
P9R2	Github	CSS	1st	✖	Request changes	2	28	446	165
P9R3	GitHub	CSS	1st	✖	Comment	2	56	842	1,177
P10R1	Github	CSS	3rd+	✖	Request changes	3	19	636	9
P10R2	Github	CSS	2nd	✖	Request changes	2	1	7	5
P10R3	Github	CSS	1st	✖	Comment	3	1	1	5

(*) Iteration number for the participant as a reviewer
(**) The code change was extremely large due to the inclusion of a large autogenerated `.yaml` file.

We observed reviews done on four online platforms (GitHub, GitLab, Gerrit, and Phabricator). We did not observe reviews in other contexts, such as IDE-integrated code review tools, reviews through emails, or in-person reviews. Therefore, our observations may not generalize to those contexts.

Since the data was collected and mainly analyzed by the first author, the results are shaped mainly by her knowledge and expertise. The second author validated the material on two separate occasions for coding and transcription quality. Furthermore, the results and interpretations were repeatedly discussed with all other co-authors in individual and group meetings and updated accordingly.

IV. RESULTS

The 25 reviews performed by ten expert reviewers led to a total of 14 hours and 42 minutes of recorded observations and follow-up interviews. The coding led to the definition of 846 codes captured in 3,792 unique references. The replication package also includes the list of themes from the analysis [53].

Our analysis showed that the Letovsky's code comprehension model [32] was efficient, yet not complete enough to cover the case of code review comprehension. Therefore, we propose an extended model to fill this gap: the Code Review Comprehension Model (CRCM), as depicted in Figure 3. Through our new model, we describe the individual components (*Code Review Process*, *Information Sources*, *Knowledge Base*, and *Mental Model*) alongside their function. While

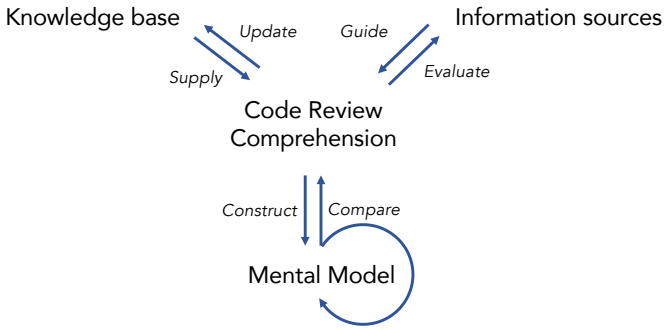


Fig. 3. Code Review Comprehension Model - Code Review Comprehension uses opportunistic strategies to enrich information sources through identifying issues, commenting, and proposing improvements.

Letovsky's model views code comprehension as combining knowledge base and information sources to form a mental model of the code, CRCM also integrates more recent code comprehension theories like constructivism [40] (*i.e.*, comprehension is a learning process that updates the knowledge base) and views code review comprehension as an opportunistic process shaped by the purpose and usage of the understanding [49] (*i.e.*, to understand code changes, select an appropriate reviewing strategy, and provide feedback). In the following subsections, we present detailed results for each component of the CRCM, their interactions, and the review scope.

A. *RQ1: Scoping Code Review Comprehension*

The goal of RQ1 is to explore the limits reviewers set for their review. We define review *comprehension scope* as the completeness and depth of the code review comprehension process. We observed reviewers employing four distinct ways of scoping their review: they performed (1) **full** reviews, *i.e.*, the reviewers achieved a complete understanding of the code changes, (2) reviews **focused** on specific aspects of the code change (*e.g.*, high-level rationale coherence and areas related to their expertise), (3) **partial** reviews of sections of the change (sometimes due to the review being terminated prematurely), and (4) **shallow** reviews where only a superficial understanding was reached, and reviewers based their judgment on external factors (*e.g.*, sufficient testing and the expertise and responsibility areas of the other involved reviewers; as P4 explained: "If the PR is really really big, I trust in the CI. I trust if all the tests are passed I understand that the changes that are being added are not affecting the current behavior.").

Reviewers reported that several factors—related to the nature of PR-based code review and code complexity—influence the completeness and thoroughness of their reviews.

Given that code review is *iterative* and *incremental* in nature, a full understanding of the code changes can be gradually achieved through multiple review iterations. Reviewers use these characteristics to break up the entire review into manageable steps and to benefit from the expectation of further review iterations. As P5 noted: "I don't need this knowledge answered right now. I need this to be answered before I approve, but I can live without it today." Each review iteration also allowed reviewers to narrow the scope of the review. When reviewing the same PR for the third time (P2R2), P2 explained: "I can take shortcuts. If [the file] has no comments, I will ignore it, I will not really review it. And this thing converges, right? Because more and more files enter the state."

Code review is also *interactive*: Reviewers and authors can support each other in understanding the code. The *collective comprehension* of the author and reviewers can have more importance than an individual reviewer reaching full comprehension of the PR. Reviewers consider the added value their review can provide, the expertise of other reviewers, and the opportunity to seek clarifications from the author. As P9 puts it: "If this was application code and I didn't understand anything, I would definitely either make sure that I understand or ask about it. Or there is another reviewer who can go a little bit deeper than me."

The complexity of the change under review fundamentally contributes to the need to scope their review comprehension. When we asked P5 why they reviewed only a part of the change, they explained: "I have to ... it is impossible to fit [it] into my head." Small changes require less comprehension or reviewer involvement; as P2 put it: "There are not many things that can go wrong in one line of code and the automated tools would have caught most of the issues already."

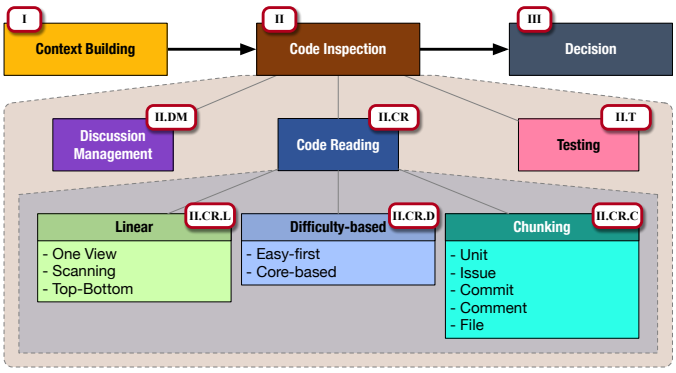
B. *RQ2: Strategies Reviewers Use To Perform Code Review*

RQ_2 explores the strategies reviewers use to understand and review the code change. The code review process, similar to code comprehension, is opportunistic. Reviewers combine various information sources with their knowledge base and existing mental models to understand the code change, evaluate the PR, and provide feedback. They deliberately select their reviewing strategy, accounting for the change's complexity, aligning with their current priorities and review scope.

Understanding in code review is reached through multiple activities, presented in Figure 4 (a) — with activities numbered and color-coded to guide the reader through the results. Figure 4 (b) uses this color scheme to represent the sequence of activities across review sessions, ordered by change size.

In most reviews (N=23), the reviewer began with a I *context building* phase, then proceeded to II *code inspection*, and concluded by submitting feedback, making a III *decision*, and potentially merging the change. During the code inspection, reviewers II.DM *manage discussions*, perform II.CR *code reading* using various strategies, and perform II.T *testing*. While reading code, we observed that reviewers tended to follow a II.CR.L *linear* reading approach for smaller changes (up to P1R3 - 6 files changed, 176 lines of code); while, for larger changes, they employed alternative strategies, such as II.CR.D *difficulty-based* reading or II.CR.C *chunking*, which enabled them to split the review into manageable units.

I Context Building. This step starts with reviewers familiarizing themselves with the PR title and description, and gathering other information sources (mentioned there or in the PR history and discussion; see Section IV-C1 for more details).



(a) Activities reviewers combine to reach an understanding and create a reviewing strategy. Each activity is described in the text using its number and colored label.

Review ID	Strategy						
P4R1	I	II.CR.L	III				
P10R3	I	II.CR.L	III				
P3R1	I	II.CR.L	III				
P10R2	I	II.CR.L	III				
P2R4	I	II.CR.L	III				
P1R2	I	II.CR.L	III				
P8R1	I	II.CR.L	III				
P7R1	I	II.CR.L	II.T	III			
P1R1	I	II.CR.L	III				
P4R2	I	II.CR.C	III				
P6R1	I	II.CR.L	II.CR.D	II.T / II.CR.D			
P7R2	I	II.CR.L	III				
P2R3	I	II.CR.L	III				
P1R3	I	II.CR.L	III				
P6R2	I	II.CR.D					
P3R2	I	II.CR.L	II.CR.D	II.CR.L	II.CR.D	III	II.CR.L / II.DM
P2R2	II.CR.L	II.CR.C	II.CR.L	II.CR.C	III		
P5R1	I	II.CR.D	II.CR.D	II.CR.C	II.DM	III	
P2R1	I	II.CR.L / II.CR.C	II.CR.L	III			
P8R2	II.DM / I	II.CR.L	III	II.DM			
P9R1	II.T / I	II.CR.D	II.CR.D	II.T	III	II.T	
P10R1	I	II.CR.D / II.CR.C	II.CR.C	II.CR.C	II.CR.C		
P9R2	II.T	II.CR.D	II.CR.D	III			
P5R2	I	II.CR.C	II.CR.C	III			
P9R3	I	II.CR.D	II.CR.D	II.CR.L / II.CR.D	II.CR.C / II.CR.D	III	

(b) Composition of these activities in each review session. The color legend corresponds to Figure 4 (a). Some activities can be employed in parallel (represented by split-color fields.) The reviews are ordered by size - smallest to largest code changes.

Fig. 4. Code Review Process Strategies

This initial step gives them an idea about the completeness of the available information and the Git hygiene [26] of the PR. It helps reviewers identify issues needing attention, actions required from them, and open questions. The context-building phase also enables reviewers to build a preliminary mental model of the PR and assess its complexity. Furthermore, reviewers form expectations about the PR and its ideal implementation, which they compare with what they see during the code inspection. In the team of P10, the PR descriptions can also explicitly include the PR acceptance criteria or suggestions for the reviewing strategy. For instance, P10R1 included a suggestion to review the PR commit by commit, which P10 followed: "I would at least give it a try without even thinking about it." Yet, the benefits of this phase are compromised when the information sources are not provided.

Reviewers reduced the need for context building during the review session by being part of creating issues before the review or holding planning and alignment meetings with their team and the change's author. In P2R2 and P9R2 the context-building phase was skipped entirely due to a high availability of the context. Five reviewers (P2, P5, P7, P8, P9, and P10) underlined the importance of having a pre-alignment with the PR's author for review efficiency. As P10 explained: "We already pre-agreed on solutions so I don't need to challenge so much. It's more like does this match my expectation of what would happen or not? So [the review] is much smoother."

II Code Inspection. This phase, shown in Figure 4(a), consists of discussion management, code reading, and testing:

II.DM Discussion Management. While reviewers mostly wrote comments while performing Code Reading, some of the reviewers allotted a specific slot in their review to read through the available discussion and interact with the comments of other reviewers. This step was used to see what other reviewers and the author have already addressed, identify areas of agreement, and identify where they might add a new perspective. As P5 put it: "maybe the discussion already explains to me that I shouldn't review it right now, because somebody already went deep into this, and my time is not even needed right now here."

II.CR Code Reading. The strategies discussed here are inspired by Heinonen et al. [24]. We observed three approaches: *linear* reading, *difficulty-based* reading, and *chunking*.

II.CR.L Linear. Reviewers use this strategy with reviews of manageable size. As P2 puts it: "There are not many things that can go wrong in one line of code". Linear reading was the main strategy used in small reviews until the review of P1R3 (176 changed LOC), as shown in Figure 4 (b).

II.CR.D Difficulty-based. Reviewers employed this strategy when review size increased and they had to deal with the growing complexity. Reviewers prioritize what to review based on reviewing difficulty. Table III summarizes examples of code that reviewers deemed difficult or easy to review. Reviewers also referred to these two categories as areas where they (do not) need to invest significant effort and energy. The difficulty-based reading appears in two forms: (1) the *easy-first* approach (P3R2, P6R1, P9R1, P9R2, P9R3) where reviewers first "get rid of easy stuff" then focus on the parts that require more effort, and (2) the *core-based* approach (P5R1, P6R2) where reviewers began with what some of them refer to as "the core of the change," then follow the data and execution flow to understand how the core changes were used in the code.

II.CR.C Chunking. This approach also deals with increasing complexity and can exist in parallel to top-bottom or difficulty-based reviewing. Reviewers may review only selected parts of a change, leaving the rest for later, or break the PR into chunks to narrow their scope and reduce their cognitive load. However, reviewing chunk by chunk can lead to a lack of overview; for example, to compensate for this

TABLE III
EXAMPLES OF CODE THAT REVIEWERS PRIORITIZED ACCORDING TO REVIEWING DIFFICULTY

Hard to review	**Examples**
Large changes	
Complex changes	Logic, chained events
Potential for substantial issues	Parallel processing
Easy to review	**Examples**
Not author-written code	Auto-generated files, boilerplate code, binary files
Not production critical	Documentation, tests
Structurally constant changes	Declarative changes, established patterns in the code
Small changes	Renaming of program entities, small logic changes, string changes, upgrades, white space changes, usage of the main implemented element
Changes with low effect on the final code quality	Unnecessary removed files, visual front-end changes, code that will change in the future

drawback, the reviewers in P2R2 and P5R2 finalized their review by linearly reading the entire change to ensure the change's consistency. Reviewers employed various units of chunking (listed in Figure 4). Those reviewing the PR commit-by-commit stressed the importance of good Git hygiene and team processes to enable an effective breakdown of the review via commits. Other units, such as functional areas – *e.g.*, models, migrations, and tests (P8R2), also served as ways to mentally segment the PR. Tests, in particular, were a significant unit of chunking. Reviewers reported different preferences as to when to review tests. P2 and P5 sometimes like to start with tests as they document the intention of the author, similarly to *test-driven review* [45]. In contrast, P8 preferred reviewing tests last. Re-reviews of the same PR present a specific scenario where reviewers use *comment-based* chunking (P2R1, P2R2, P5R2) to selectively check how were their previous comments addressed by the author. Furthermore, we observed reviewers chunking their review based on *files, issues, etc.*

II.T Testing. During code inspection reviewers not only review tests but also actively *test* the PR (P6R1, P7R1, P9R1 and P9R2). They check the expected and actual output of functions in their local terminal, verify whether the system runs properly in their local environment with the new changes, perform hands-on testing of the responsiveness of the implemented UI changes, or troubleshoot failed CI/CD checks.

III Decision. Reviewers submit their comments, give overall feedback, and provide a verdict: accept the change, leave comments, or request further changes. They finalize the review once they have checked all necessary parts of the PR, reached a desired level of understanding, provided sufficient feedback, know that they have no pending notes, and wrapped up any ongoing discussions.

C. *RQ3*: Role of information sources, knowledge base, and mental models

Code review comprehension strategies rely on the information sources that include the reviewed code change and the reviewer's knowledge to construct a mental model of the code change. The mental models are then used to update the knowledge base, compare with other mental models (such as mental models of expected and ideal solutions), update them, and consequently evaluate the code change and provide feedback. In the following section, we provide an overview of the role of these CRCM components in review comprehension.

1) Information Sources: Reviewers use many information sources during code review. The ones explicitly mentioned in the most reviews are the PR title and description (21 reviews out of 25), the issue tracking (11 reviews), and the PR discussion (10 reviews). Reviewers use (1) information sources linked within the PR itself (*e.g.*, review size, commit titles, or CI/CD status), (2) resources to understand the broader code context of the PR (*e.g.*, code base, tests, or documentation), (3) tools to evaluate and test the change (*e.g.*, local development environment, specialized software tools), and (4) external sources not directly connected to the software system (*e.g.*, language documentation, ChatGPT, blog posts). P7 and P9 both used specialized code review applications developed within their companies to *test* the system behavior after a patch is applied and navigate the code base history.

We classified in which review stage reviewers used different resources. During the *context-building* phase, which occurs at the start of the review, most of the information, especially what is presented and linked within the PR, is gathered. Interestingly, in 20% of the observed sessions, reviewers needed to refer to other PRs for more context. While PR discussion is accessed throughout the review, specific activities rely on different information sources. For instance, in the code inspection phase, reviewers use resources that help them navigate the code base, evaluate whether the PR was implemented correctly, and test the PR. Reviewers tested the PR in their local instance of the software system and used other specialized tools. Before making the final verdict, some reviewers revisited information sources such as CI/CD status or to-do notes.

2) Knowledge Base: This component of the model plays a key role in code review comprehension and in directly evaluating the PR for potential issues. Letovsky's model of code comprehension provides a broad depiction of knowledge that understanders use: from high-level domain knowledge and programming plans to the knowledge of programming language semantics. These fundamental aspects of code comprehension enable reviewers to recognize the 'correct' implementation of the domain concepts, engineering processes, programming goals, and code formatting, and to set an expectation for code quality [46]. We observed that reviewers also employ their knowledge gained through their own experience as software developers, acquired through their interactions with code and other developers. This knowledge ranges from technical experience and familiarity with tools to their strategies to solve programming problems, to knowledge of their colleagues' expertise, coding and working style, and learning needs. Reviewers use this knowledge to provide more focused feedback: "If you don't have specific [coding] rules because

you are a junior developer I will help you create them" (P6).

The more familiar a reviewer is with the overall context of the PR, the fewer resources they need to understand the PR and related artifacts. This helped enhance their ability to give constructive feedback for improvements. As mentioned in the *context-building* phase, activities such as issue writing or pre-alignment with the team before the review reduce the need to build the context from information sources within the review itself. For instance, P1 quickly navigated code and provided feedback without stopping to understand or think further. When asked about it, they explained that it was due to their role in gatekeeping all code changes in the project and their extensive knowledge of the code base. They further clarified: "I have been reviewing code for a while, but there will be always changes that are completely new. And in those cases, it will take me longer. It could take even an hour to go through the test, match it with what I know, and figure it out". The knowledge base also contains *efficiency knowledge* [32], which refers to the explicit knowledge of common issues that can be directly applied to identify them. This shows the fundamental role the knowledge base plays in context building and improving the efficiency and thoroughness of code reviews.

Below we report on the mental models reviewers construct during their review sessions. The knowledge base may already contain mental models stored in long-term memory that were constructed prior to the review and that can be used during a review session: (1) a mental model of the software system—keeping knowledge of the processes, standards, expectations, and coding patterns in the code base that allows them to identify inconsistency with the system architecture, company and team coding practices; and (2) a mental model of the PR—reviewers may already have enough context about the PR to form a preliminary mental model of it.

The mental models of the system and the PR are constructed and updated through review iterations and stored in long-term memory for future recall. Thus, there is an ongoing exchange between the mental models in the knowledge base and the ones resulting from the comprehension process in the review session itself.

3) Mental Models: A mental model is the developer's mental representation of a program or code entity [24]—the PR in code review context. In Letovsky's representation [32], the mental model is constructed in three layers: the specification, the implementation, and the annotation layer. Here, we present in detail (1) the three layers of the mental model of the PR and (2) the use of alternative mental models of the PR as expectations and ideals that can be used for comprehension as well as PR evaluation.

Layers of the Mental Model of the PR: An essential part of constructing the mental model of the PR is establishing the modification and increment to the software system—what parts of the system were changed and the size and complexity of these changes. Reviewers achieve this by using the available information sources (Section IV-C1) and by understanding the code itself (*e.g.*, by comparing the removed and added parts of the code, code tracing, or observing the system's behavior with the changes applied). This overview aids them in populating all three layers of their mental model.

Specification refers to an explicit, complete description of the program's goals [32]. Reviewers need to understand the review goal, *i.e.*, the problem being solved, the reasons for the change, and the scope of addressed cases. As they construct the specifications of the mental model of the PR, they also set explicit expectations for what the implementation should include, such as which tasks and code units the PR may contain, thus defining the evaluation criteria for the implementation. For instance, in the review *P5R1*, the reviewer reads the title of a PR that introduces a counter for lazy loads on a specific object. The reviewer anticipates that the author probably counts lazy loads per request and logs them, which aligns with the actual implementation found later in the review.

The *implementation* layer of the mental model refers to the actions and data structures in the program [32]. Reviewers need to understand what has changed in the implementation and use code tracing [39] to infer the code's behavior and its output. They also need to assess the rationale behind implementation choices and whether the changes are located correctly in the code. Reviewers in the first review iterations on the PR might focus on ensuring the overall logic is sound before they dive into the implementation details. For instance, P10 explained: "I was thinking if it needs some other mapped table ... but that doesn't matter that much as it is a technical thing and I am now checking mostly whether it makes sense".

The *annotation* layer connects specification goals to the parts of the implementation that fulfill them and which parts of the implementation fulfill certain specifications [32]. As suggested, the annotation is done by reviewers by using top-down and bottom-up processes employed in parallel. When reviewers build their mental model in the context-building phase of the review, they can already construct all three layers. Many information sources allow them to build specifications and create expectations that are merely confirmed in a top-down manner during code inspection. The top-down annotation is supported by meaningful traceability across the issue tracking, PR title, and description, commit messages, file and variable naming to the implementation layer or by tracing review comments to their fixes. The bottom-up annotation starts by understanding code behavior and then assigning purpose to it. Reviewers commonly comprehended a piece of the implementation and noted to themselves that it 'made sense'(P2, P3, P7, P8, P9, P10).

These three layers of the mental model create a complete understanding. Therefore, the review process can be streamlined by supplying reviewers early with cues that help them identify the change's goals and annotate them to expectations on how these goals are implemented. The expected version of the PR might, however, prove to be different from the actual PR implementation and specification during the review.

The Expected and the Ideal: Reviewers form alternatives, variants, and extensions of their mental model of the PR in the form of expected and ideal solutions.

As mentioned in Sections IV-C2 and IV-C3, reviewers tend to form expectations about the PR, which can reach the details of a partially formed mental model. We call this mental model the *expected* mental model. Such a pre-existing model only needs to be confirmed when reviewing the PR. These expectations can stem from various sources, such as previous review rounds and entered comments, pre-alignment on solutions with the author, or general software engineering practices. Adhering to the expectations is desirable in the eyes of the reviewer (P8: "This is pretty much what I expected to see. So I'll just approve it".) and makes the review more efficient (P10: "We already pre-agreed on solutions. So ... it's more like: Does this match my expectation of what would happen or not?")

An alternative set of models are the *ideal* mental models, representing the optimal, corrected, or improved version of the PR. P7 remarked: "Now I'm wondering what the ideal solution is...". These models are supported by the reviewer's programming experience, personal preferences, and knowledge of good engineering practices. Adherence to this imagined optimum is valued but not necessarily followed. P10 puts it: "In the end I don't necessarily agree that this is a better solution, but it still somehow solves the problem and it doesn't add a technical debt or anything like that".

Both the expected and the ideal mental models offer an alternative or extension to the mental model of the actual PR. They may not always be formed or used, but when they are, they can be a vital tool to perform the review. Overall, mental models help reviewers set their expectations and evaluate changed codes against them, thus streamlining the review and identifying areas of potential improvements.

V. Discussion

In this study, we observed 25 review sessions to analyze reviewing strategies through the lens of code comprehension. Extending Letovsky's model of code comprehension [32], we developed the Code Review Comprehension Model, detailing how its components – the information sources, knowledge base, and mental models interact and shape code review.

A. Findings

Code comprehension is central to code review. Reviewers' ability to perform code review is dependent on their ability to understand the code change [52, 2, 19, 33]. Letovsky's model [32] offers a valuable framework to explain how comprehension shapes review, because reaching an understanding is among the reviewers' criteria to finalize their review and accept changes and drives them to seek information sources and choose effective reviewing strategies. Using the constructivist perspective on code comprehension [40], we described how reviewers update their knowledge of programming practices and the software system through reviewing code. By viewing code comprehension as a tool to perform software development tasks, such as in the work of Von Mayrhauser and Vans, we also described how the comprehension process contributes to evaluating the code changes and providing feedback.

Code review comprehension is scoped. According to Letovsky [32], full understanding is achieved when all three layers of a mental model are fully developed. In practice, full understanding can be reached over the entire PR lifetime, rather than in a single review session. As predicted by Piaget's theory of cognitive development [38], we observed that developers scope their reviewing and understanding. Full comprehension may not always be the goal, especially when reviewers perform focused, partial, or shallow reviews. Scoping down the review is a commonly used strategy to deal with review complexity—one of the main challenges reviewers face [29].

Code review comprehension is incremental, iterative, and interactive. Reviewers already have mental models of the software system and may already have one for the PR itself before starting their review sessions. This knowledge allows them to be efficient when reviewing the PR. Reviewers' mental model of the PR develops over time through review iterations and is used to update their mental model of the software system. Importantly, reviewers interact with the author and other reviewers to reach the desired level of understanding.

Reviewing is strategic. Ad hoc reviewing is considered unsystematic [12, 48]. However, we have observed that reviewers employ opportunistic strategies, similar to code comprehension, to reach an understanding of the PR and review the change. Reviewers combine several activities in a modular way to form their reviewing strategy. First, they build the context. Then, they combine code reading, discussion management, and testing to comprehend and evaluate the change. The scoping and the modular design of their reviewing strategy allow them to creatively deal with change complexity.

Ideals and expectations. Reviewers approach the review with ideas on ideal and optimal solutions and other expectations towards the PR, informed by common engineering solutions, good practices, and their knowledge of the PR as well as the software system. Using the self-discrepancy theory [3], we could interpret the data effectively, identifying developers alternative mental models of the ideal and expected code changes. Meeting these ideals and expectations is seen as correct and streamlines the review by aiding reviewers to efficiently interpret the code, give feedback, suggest alternatives, and identify issues.

B. Implications and Recommendations

Based on these findings, we discuss recommendations on how to support reviewers in effective code review comprehension and change evaluation.

For Code Review Tools: Our results provide insights for designing code review tools that better align with reviewers' natural strategies. First, reviewers scope their review sessions and prioritize their activities. Tools can support reviewers in viewing only certain aspects and parts of the PR, as well as provide ways to let reviewers communicate the scope and main outcomes of the performed review.

The main strategies for code reading while dealing with review complexity were chunking and difficulty-based reading.

This finding is an opportunity to investigate how tools can provide support to identify and visually separate meaningful code chunks, indicate the change distribution by signifying the core of the change or more complex passages, and support Git hygiene. Reviewers often switched contexts to navigate the code base or test the code change. Both these activities were substituted in two cases (P7, P9) by an in-house reviewing application. This behavior shows there is a need to integrate code base navigation, testing, and reviewing in one environment together with information sources, such as issue trackers or online/documentation search.

Reviewers used review comments as an information source throughout the review, while remaining wary that comments could be a source of potential bias. Tools should be designed to allow reviewers to be able to toggle the viewing of existing review comments. Furthermore, reviewers created many mental models of the code change, their expectations, and ideals. Tools can support developers to collaboratively construct and cross-reference multiple mental models [47].

Future research can be designed and carried out to investigate whether and how AI and LLMs have the potential to support many of the aforementioned improvements, *e.g.*, by suggesting effective reviewing strategies, annotating review comments to related fixes in the implementation, or generating missing context and documentation [42].

For Practitioners: We observed real-world code reviews performed by experienced reviewers recommended by others as great reviewers—their practices can be used to propose effective reviewing strategies.

For *dealing with review complexity*, our participants mainly used three strategies: (1) narrowing down the scope of the review, (2) using iterations to reach full understanding one step at a time, and (3) adjusting the reading approach (*e.g.*, using chunking). The effectiveness of the first strategy is reinforced by previous research, which found that focused reviews enhance reviewers' ability to detect defects [9]. To better support the third strategy, good Git hygiene is helpful; and, if there is a clear core of the change, it can be used as a starting point for reading.

For *improving reviewing efficiency*, our participants relied on a strong knowledge base, by knowing programming practices, the codebase, and the team. Additionally, they also relied on the colleagues involved in the review process to complement their expertise and understanding. Finally, creating expectations about the change under review and having an alignment with the change author prior to the review session was used to improve reviewing effectiveness.

For Researchers: Code comprehension is a stepping stone in performing software development tasks [50]. Letovsky's model of code comprehension has proved to be good basis to start describing how code review is performed in a real scenario, including the diversity of reviewers' strategies and information sources. Using this model can be applied to other software development tasks to formulate recommendations towards more human-centric tool design.

Reviewers' activity has been previously approached with methods such as experiments [22, 21, 18], interviews [45], eye-tracking [43], and mining software repositories [9, 36, 21]. However, these methods have limitations in capturing critical factors such as context, priorities, interactions among reviewers, and their knowledge of the software system being reviewed. Usually, eye-tracking studies and experiments use artificial code changes of a manageable size where linear reviewing strategies were mostly observed [43, 21]. However, we have observed that complex reviews in real-world context require strategies to manage the complexity. These strategies include narrowing the scope of the review or concluding it when the reviewers feel their comments no longer provide significant value, or when they run out of time or energy to continue the review. This behavior may explain why Fregnan et al. [21] observed a linear decrease of the number of comments in each subsequent file of the PR or contextualize the test-driven review among other reviewing practices [45].

For Educators: Students benefit from being taught (1) explicit programming strategies and (2) to reflect on their practices [28]. Our findings can aid students to reflect on their reviewing strategies. Moreover, students can be instructed on potential reviewing strategies, decomposing PRs in more reviewable units, and finding appropriate information resources. Students can learn to leverage the experience and expertise of other developers and create rich expectations for change evaluation through building rich context and documentation of PRs and creating alignment with their teammates. Educators should emphasize the importance of a strong foundation in programming patterns to enhance the effectiveness and efficiency of code reviews. This is particularly crucial as the use of AI-generated code becomes more prevalent, requiring students to develop the skills needed to review such code effectively.

VI. Conclusion

In the study presented in this paper, by observing expert reviewers—recommended as great reviewers by others—while reviewing real-world code changes and interviewing them afterwards, we could uncover, for the first time, the opportunistic strategies they use to form an understanding of the code change and to evaluate it. Based on these strategies, we developed a Code Review Comprehension Model and put forward recommendations on how to improve code review tools as well as on topics to conduct further research.

Acknowledgment

Alberto Bacchelli and Pooja Rani gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects 200021_197227 and 200021M_205146. Pavlína Wurzel Gonçalves gratefully acknowledges the support of CHOOSE, the Swiss Group for Original and Outside-the-box Software Engineering (https://choose.swissinformatics.org/).

References

[1] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, 48 (12):4925–4946, 2021.

[2] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[3] Waclaw Bak. Self-standards and self-discrepancies. a structural model of self-knowledge. *Current Psychology*, 33(2):155–173, 2014.

[4] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022.

[5] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors influencing code review processes in industry. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 85–96, 2016.

[6] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 329–340. IEEE, 2017.

[7] Deborah A Boehm-Davis, Robert W Holt, and Alan C Schultz. The role of program structure in software maintenance. *International Journal of Man-Machine Studies*, 36(1):21–63, 1992.

[8] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[9] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

[10] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737–751, 1977.

[11] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 557–566, 2007.

[12] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews, the state of the practice. *IEEE software*, 20(6):46–51, 2003.

[13] Thomas A Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[14] John W Creswell. *Educational research: Planning, conducting, and evaluating quantitative and qualitative research*. pearson, 2015.

[15] John W Creswell and Dana L Miller. Determining validity in qualitative inquiry. *Theory into practice*, 39 (3):124–130, 2000.

[16] Nicole Davila and Ingrid Nunes. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:110951, 2021.

[17] Gordon Diaper. The hawthorne effect: A fresh examination. *Educational studies*, 16(3):261–267, 1990.

[18] Alastair Dunsmore, Marc Roper, and Murray Wood. The role of comprehension in software inspection. *Journal of Systems and Software*, 52(2-3):121–129, 2000.

[19] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38 (2.3):258–287, 1999.

[20] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.

[21] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalıklı, and Alberto Bacchelli. 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*, pages 483–494, 2022.

[22] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do explicit review strategies improve code review performance? towards understanding the role of cognitive load. *Empirical Software Engineering*, 27(4):99, 2022.

[23] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference on software engineering*, pages 345–355, 2014.

[24] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. Synthesizing research on programmers' mental models of programs, tasks and concepts—a systematic literature review. *Information and Software Technology*, page 107300, 2023.

[25] Monique M Hennink, Bonnie N Kaiser, and Vincent C Marconi. Code saturation versus meaning saturation: how many interviews are enough? *Qualitative health research*, 27(4):591–608, 2017.

[26] Nick Hodges. Six rules for good git hygiene, 2019. URL https://betterprogramming.pub/six-rules-for-good-git-hygiene-5006cf9e9e2.

[27] Robin Jeffries. A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of AERA annual meeting*, volume 10, pages 1–7, 1982.

[28] Amy J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. Teaching explicit programming strategies to adolescents. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 469–475, 2019.

[29] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: How developers see it. In

Proceedings of the 38th international conference on software engineering, pages 1028–1038, 2016.

[30] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. Explicit programming strategies. *Empirical Software Engineering*, 25(4):2416–2449, 2020.

[31] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2010.

[32] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[33] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(4):34–42, 2017.

[34] Justin Middleton and Kathryn T Stolee. Understanding similar code through comparative comprehension. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11. IEEE, 2022.

[35] Andy Oram and Greg Wilson. *Making software: What really works, and why we believe it*. " O'Reilly Media, Inc.", 2010.

[36] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–27, 2018.

[37] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[38] Jean Piaget. Cognitive development in children: Piaget development and learning. *Journal, of Research in Science Teaching*, 2:176–186, 1964.

[39] Ruixiang Qi and Davide Fossati. Unlimited trace tutor: Learning code tracing with automatically generated programs. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 427–433, 2020.

[40] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*, pages 271–278. IEEE, 2002.

[41] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at Google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190, 2018.

[42] Md Nazmus Sakib, Md Athikul Islam, and Md Mashrur Arifin. Automatic pull request description generation using llms: A T5 model approach. *arXiv preprint arXiv:2408.00921*, 2024.

[43] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the symposium on eye tracking research and applications*, pages 381–384, 2012.

[44] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5):595–609, 1984.

[45] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-driven code review: an empirical study. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1061–1072. IEEE, 2019.

[46] Diomidis Spinellis, Panos Louridas, Maria Kechagia, and Tushar Sharma. Broken windows: Exploring the applicability of a controversial theory on code quality. In *Proceedings of 40th International Conference on Software Maintenance and Evolution*, ICSME '24, New York, NY, USA, 2024. IEEE. doi: 10.1145/3643665.3648048.

[47] M-AD Storey, F David Fracchia, and Hausi A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[48] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140, 2006.

[49] Anneliese Von Mayrhauser and A Marie Vans. From program comprehension to tool requirements for an industrial environment. In *[1993] IEEE Second Workshop on Program Comprehension*, pages 78–86. IEEE, 1993.

[50] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, aug 1995. ISSN 0018-9162. doi: 10.1109/2.402076. URL https://doi.org/10.1109/2.402076.

[51] T Winograd and DD Woods. The challenge of human-centered design. *Human-centered systems: information, interactivity, and intelligence*, 1997.

[52] Pavlína Wurzel Gonçalves, Gül Calikli, Alexander Serebrenik, and Alberto Bacchelli. Competencies for code review. *Proceedings of the ACM on Human-Computer Interaction*, 7(CSCW1):1–33, 2023.

[53] Pavlina Wurzel Goncalves, Pooja Rani, Diomidis Spinellis, Margareth Storey-Anne, and Alberto Bacchelli. Replication package for 'code review comprehension: Reviewing strategies seen through code comprehension theories', 2025. URL https://zenodo.org/records/14748996.