

Code Comprehension Strategies for AI-Assisted Development

Uroš Kosič

University of Ljubljana
Kongresni trg 12, 1000 Ljubljana
ukosic@gmail.com

1 Introduction

Software developers routinely work with codebases designed by others - colleagues, predecessors, or, increasingly, AI assistants. Before extending or refactoring such a codebase, they must reconstruct a mental model of how it operates and why it is shaped that way. Naur (1985) argues that source code is only a partial expression of the developer's underlying theory of the system; the rest lives in the developer's mind. Comprehension is therefore an active process: developers read code, run experiments to test hypotheses, and externalise structure into sketches to reduce cognitive load. Current AI workflows can substitute for this cognitive work when used uncritically, but whether and how they do so remains an open question. Despite a long tradition of program comprehension research, the strategies that distinguish accurate, efficient comprehension of unfamiliar code remain poorly catalogued, and the comprehension literature largely predates the LLM era, leaving open how AI-assisted workflows should be designed around them.

2 Purpose

The primary contribution of this study is a taxonomy of comprehension strategies experienced developers employ on unfamiliar codebases. As a secondary contribution, we use the taxonomy to derive design principles for AI-assisted workflows that scaffold, rather than displace, the strategies it identifies. We hypothesise that (a) comprehension is dominated by active engagement over passive reading; (b) IDE-mediated structural navigation differentiates effective developers; (c) successful developers iterate rapidly between hypothesis and empirical test; and (d) flexible switching between strategies is itself a meta-strategy associated with success, while persistence in a single strategy is associated with longer task time and lower correctness.

3 Methods

We propose a mixed-methods, primarily qualitative design followed by an analytic step that uses the resulting taxonomy as a lens for comparative analysis of existing AI workflows. Ten experienced developers (5 women, 5 men; ≥ 10 years professional experience, at least half in Java, microservices and Docker) each attempt two tasks on a medium complexity Java/Quarkus/Docker codebase: (a) adding a feature requiring comprehension of business logic, and (b) locating a performance

bug. The study runs in two sessions: an initial interview with demographic questionnaire, then a task session. Screen recording and a structured observation sheet capture on-screen actions, followed by a retrospective interview and a whiteboard architecture sketch that externalises the participant's mental model for validation. A two-participant pilot will verify tasks, timing, and protocols. Recordings, observation sheets, and interview transcripts are inductively coded through thematic analysis to produce the taxonomy. The taxonomy is then mapped against representative AI workflows (chat-based assistants, inline autocomplete, and agentic IDE tools), to identify where each supports or displaces effective strategies, yielding design principles that preserve theory building.

4 Expected Results

We expect to identify a set of first-order comprehension strategies. The two tasks should elicit distinct profiles: feature addition foregrounding domain-understanding strategies, and performance debugging foregrounding hypothesis-driven technical investigation. Mapped against AI workflows, we expect the majority to provide no explicit space for constructing the developer's model of the system.

5 Implications

The taxonomy offers a concrete account of what effective code comprehension looks like in practice. Mapped against AI workflows, it points to design changes that help AI tools support the developer's mental-model building rather than replace it, and gives practitioners a basis for deciding when AI assistance helps and when it undermines understanding.

References

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253–261.