Code Explanations: Automated Hierarchical Descriptions of Program Behavior

Chetan Goenka
cgoenka@berkeley.edu
UC Berkeley EECS
Berkeley, California, USA

J.D. Zamfirescu-Pereira
zamfi@berkeley.edu
UC Berkeley EECS
Berkeley, California, USA

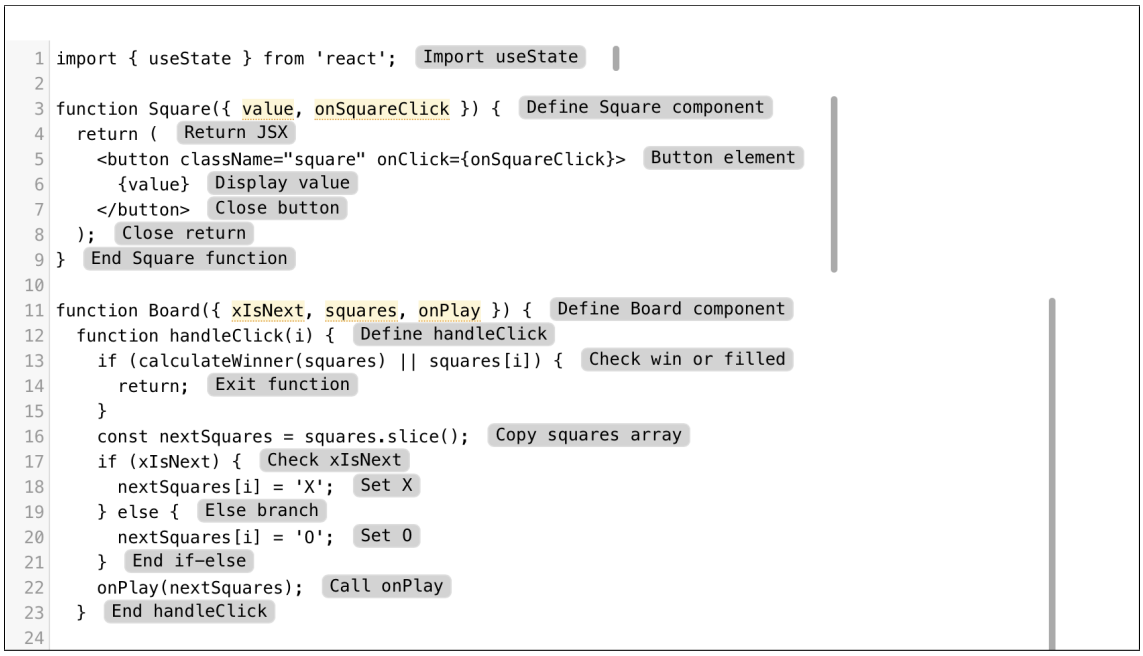

Figure 1: Overview of Code Explanations Interface

Abstract

With the increase in LLM-generated code used both personally and professionally, the need to help users understand how computer programs—a quintessential tool for thought—operate, and how they might be modified, is more urgent than ever. This workshop paper presents one approach at automatically generating explanations for a program's behavior at three levels of abstraction: per-line, per-block, and by function parameter. Our hope is that these explanations can help users make better sense of AI-generated code and understand how to make changes towards a specific goal.

Keywords

LLM Code Generation; Explanations of Code

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Chetan Goenka and J.D. Zamfirescu-Pereira. 2025. Code Explanations: Automated Hierarchical Descriptions of Program Behavior. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

LLM-based code generation is becoming an increasingly popular way to author computer programs, for both professional and personal use. Professionally, it has been reported taht over 25% of code at AI and search giant Google is reportedly authored by AI [1]. Personally, new software platforms like OpenAI's *Canvas*[1] and Anthropic's Claude *Artifacts*[2] enable the prototyping of software by non-experts with no prior software expertise. These latter examples are often pitched as tools for thought in the sense that they enable the creation of personalized software in service of some personal epistemic goal. Understanding code generated by AI is thus an increasingly urgent challenge, especially because the population

[1] https://openai.com/index/introducing-canvas/
[2] https://support.anthropic.com/en/articles/9487310-what-are-artifacts-and-how-do-i-use-them

needing to do so may skew increasingly towards non-experts, that is, people with only a few years' experience, if any at all.

In this workshop paper, we present CodeExplainer, a tool that shows hierarchical high-level explanations for code, at three levels:

(1) Individual lines of code
(2) Blocks of code, such as functions, classes, etc.
(3) Data flow: variables, parameters, and other state that is transformed and passed through functions and classes.

CodeExplainer displays these explanations *in situ* as an interactive overlay, as shown in Figure 1. Two-word summaries of the explanation for each line are shown persistently, while hovering over these summaries expands a full explanation for the given line. Block-level explanations are revealed on mouse hover over a bar spanning the block itself; data flow explanations are revealed on mouse hover over a function parameter, including destructured parameters. These explanations are all generated by prompts we designed using GPT-4o.

Through a small pilot, we find although explanations have the potential to be useful, their usefulness is highly context- and task-specific, and more work is needed to understand how best to generate and present code explanations.

2 Background

LLM-based code generation models have seen an explosion of interest, driven in part by benchmarks like SWE-bench [5] for assessing modle performance on a battery of real-world tasks. Multi-agent models like MetaGPT [4] and CHATDEV [8], and computer-controlling software engineering models like SWE-agent [9], suggest a future in which more and more code and software is generated by AI.

There has also been a lot of interest in HCI about how people use these models; Barke *et al.* [2] show that experts and novices use these differently, while Vaithilingam *et al.* show in *Expectation vs. experience* that these models are not all just increasing in success rates over time, but indeed all come with drawbacks as well.

Most recently, Yen *et al.* [10] use natural language descriptions in code synthesis through "prompt blocks" to give programmers more control, but this still operates with an assumption that programmers are authoring code directly. In the end-user programming context, Liu *et al.* [7] propose generating explanations in "naturalistic" language that is specifically grounded in LLM-generated code.

In contrast, not much work has focused on how people attempt to understand the code that is generated by these models; that's what we attempt to do here.

Before the advent of LLMs, tools to help users understand code behavior tended to target more-novice users, such as Ko & Meyers' *Whyline* [6] debugger. Whyline produces user-guided counterfactuals ("Why did..." or "Why didn't..." something happen?) to support users' understanding of the behaviors of programs.

Lastly, Code Explanations also build on prior work visualizing program code in a user-controllable hierarchy, including CodeBubbles [3].

3 Code Explanations: Goals & Design

We grounded our inquiry by identifying a set of questions users might ask about a set of code upon encountering it for the first time: [cite program understanding literature here] 1) How is this code structured, or modularized? 2) What are the individual primitives used? 3) How is data represented, and how is it shared among the code's structures? 4) What do individual lines of code do—how do they contribute to their enclosing hierarchical structure?

Drawing on this prior work and our personal experiences, we landed on three hierarchical levels at which we wanted to help users understand a particular body of code:

HL1 Individual lines of code, as a low-level basis.
HL2 Higher-level components (e.g., functions, classes) that comprise those individual lines.
HL3 Interconnections between these components, in particular, how data is shared and flows between them.

We chose these three levels as a set of approachable explanations that could provide helpful insights for users across a wider set of experience levels. Novice programmers, or programmers working in an unfamiliar language, might find explanations of individual lines of code helpful where their purpose is not otherwise clear; experts, similarly, might find the higher-level explanations help ground their exploration of a body of code while searching for where some particular functionality is implemented.

One challenge in producing and displaying explanations using this hierarchy is that there is potentially a lot of data to show all at once, especially at level HL1. We address this challenge by adding an additional information hierarchy for the by-line explanations, described later.

4 Code Explanations: Implementation

Code Explanations consists of three primary components:

(1) A CodeMirror-based code editor
(2) An LLM-powered explanation generator
(3) Interactive annotation overlays for explanations

These components work together to provide context-aware insights into the workings of the code at three different levels: Line-by-Line, Block-wise, and Function Parameter Usage.

We use the Zod library to create a structured prompt query for OpenAI's GPT-4o model. This structured output provides the necessary details, such as explanations, line numbers, and function parameters, ensuring that the explanation elements are correctly positioned and rendered with the corresponding content. The client side implementation is written in JavaScript using the React framework. We extend CodeMirror's annotation capabilities by adding DOM-based annotations to overlay interactive elements onto the code editor. The system's interactivity is driven by event-based actions such as hovering and clicking, which expand inline tooltips to reveal full explanation texts.

4.1 Line-by-Line Explanations

Line-by-Line explanations offer low-level insights into the code through overlays and inline tooltips. with a 2-4 word summary that captures the essence of the line's function. This concise description is overlaid on the code editor by default. Users can hover over these

summaries to expand the annotation element and reveal a more detailed explanation for the corresponding line of code.

4.2 Block-Wise Explanations

Block-level explanations provide high-level insights into logical sections of the code. We allow the LLM to determine what constitutes a block. Each block explanation contains the following key details:

(1) A summary of the block's purpose
(2) Inputs: The key data, variables, or components used
(3) Outputs: The results or changes produced
(4) Process: A concise description of the steps taken to achieve its functionality

Each block is visually represented by a vertical line overlay on the code editor, marking the block's start and end. When users hover over this vertical line, the corresponding block explanation is displayed as a tooltip.

4.3 Function Parameter Explanations

Function parameter explanations provide insights into function parameters and their occurrences throughout the code. Parameters in function definitions are dynamically linked to all their occurrences in the code. Each explanation includes a summary of the function parameter's purpose and usage, and for every occurrence of a function parameter, a context-aware explanation is generated.

When users hover over a highlighted function parameter, its explanation is displayed on screen. Clicking a function parameter highlights all its occurrences and displays their explanations. In this state, hovering over an occurrence temporarily hides its explanation to improve code visibility and reduce tooltip overlap. Clicking the function parameter again restores the original view, removing the expanded explanations.

5 Pilot Evaluation

We ran a small pilot evaluation with two members of our lab group, both Computer Science PhD students. Each participant was tasked with extending a tic-tac-toe game, written in JavaScript using React, to add a "highlight" on the winning line. This code was chosen from an official React documentation example. One participant was an expert in React, while the other, though familiar, was not an expert. Our goal was to identify how useful the explanations were to understanding code and aspects that can be improved. We were also interested in tracking the difference in usage between the hierarchical levels of explanations—line-by-line, block-level, and function parameter usage.

5.1 Explanation Content and Levels

Both participants found explanations most useful when they provided high-level insights rather than syntactical or logical descriptions of individual lines. This was especially true for simple lines of code, such as those containing parentheses or HTML tags. One participant suggested that these explanations could instead convey higher-level context, such as indicating which function or class is being closed. The same participant also noted that when trying to understand low-level details, they preferred "thinking in code rather than English," suggesting that explanations may sometimes disrupt their thought process. Another key observation was that explanations were mostly used when participants did not immediately understand something from the code itself, rather than as a way to gain extra information or context. A participant, who was asked to read the block-level explanations before starting the task, noted that they were "helpful to read at the start for framing of the problem," reinforcing this insight. Additionally, the participant found a specific set of line explanations particularly useful, as they clarified the purpose of an array containing indices for possible winning combinations—explaining whether a group corresponded to the top, middle, or bottom row, the left, middle, or right column, or a diagonal. This suggests that explanations may be more useful when they are tailored to provide higher level insights and context-aware details about function and variable usage..

5.2 Data Flow and Function Level Explanations

A key point in the feedback was the need for function-level explanations as a midpoint between low-level line explanations and high-level block explanations. An important observation was that the task required users to trace function call flows and data usage, which they had to do manually by searching the code for specific function or variable names to locate their uses. One participant noted that explanations aiding in understanding and visualizing function call flows could be very beneficial. Further, while working on the task, the participant encountered a particular line explanation and remarked, "Well, that wasn't useful. I want to know where this variable is being updated." These observations suggest that beyond static, text-based explanations, features allowing users to trace function calls and variable usage would be valuable additions.

6 Future Work

Our pilot study provided valuable insights into how hierarchical explanations aid code comprehension, areas for improvement, and potential new features. To gain deeper insights, a next step for us would be to conduct a more thorough evaluation with participants spanning a wide range of coding experience, from beginners to experts. This would help us understand how familiarity with programming concepts and debugging strategies influences the use of explanations and preferences for their depth and content.

An important area for future work is improving explanation content based on the feedback we received in our study. This includes focusing on function calls and data usage to help users better map out the flow of control in a program. Moreover, expanding the hierarchy of explanations—such as introducing function-level explanations and other higher-level options—could further support user comprehension. Future iterations of the system could also add adaptive explanations that adjust to user preferences or highlight different aspects of the code based on the user's needs at a given moment.

Adding interactive features could further enhance usability. For example, allowing users to trace function calls and variable updates could reduce the effort of tracking these manually. Additionally, collapsible code sections that toggle between high-level explanations and detailed breakdowns may provide a more flexible experience.

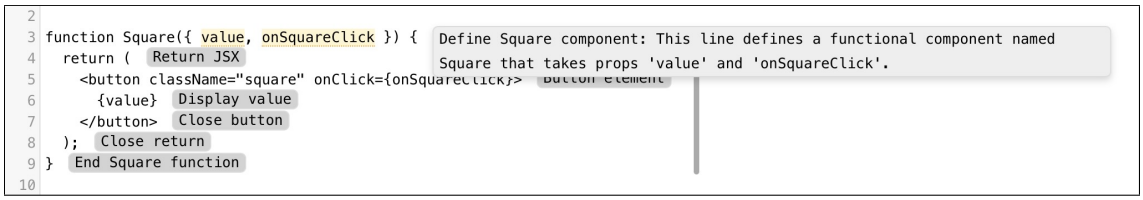

Figure 2: Line Explanation Example

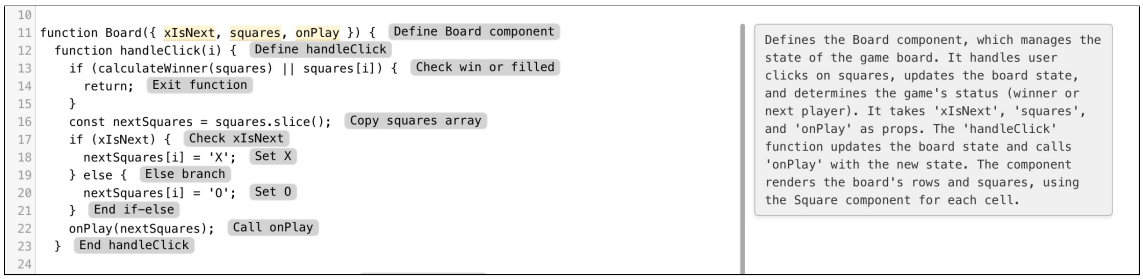

Figure 3: Block Explanation Example

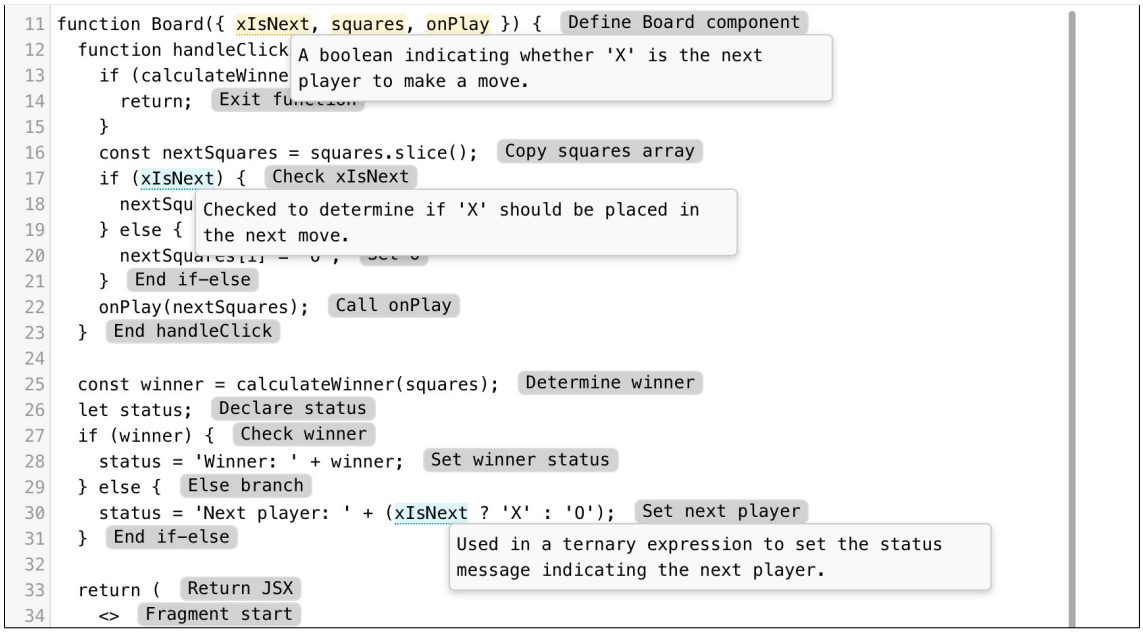

Figure 4: Function Parameter Explanation Example

Finally, evaluating the tool in different contexts, such as debugging, onboarding to a new codebase, or interview preparation, would offer insight into its broader applicability. Testing the system across different programming languages could also help determine whether the explanation strategies generalize well beyond the JavaScript-React environment used in our study.

References

[1] 2024. Q3 earnings call: CEO's remarks. https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024/

[2] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111. Publisher: ACM New York, NY, USA.

[3] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 455–464. doi:10.1145/1806799.1806866 ISSN: 1558-1225.

[4] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023.

MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. https://openreview.net/forum?id=VtmBAGCN7o

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? doi:10.48550/arXiv.2310.06770 arXiv:2310.06770 [cs].

[6] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/985692.985712

[7] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. 1–31.

[8] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. doi:10.48550/arXiv.2307.07924 arXiv:2307.07924 [cs].

[9] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. doi:10.48550/arXiv.2405.15793 arXiv:2405.15793 [cs].

[10] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, Pittsburgh PA USA, 1–20. doi:10.1145/3654777.3676357