CARNEGIE MELLON UNIVERSITY

DOCTORAL THESIS PROPOSAL

Meta-Information to Support Sensemaking by Developers

Author:
Amber HORVATH

Supervisor:
Dr. Brad A. MYERS

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Human-Computer Interaction Institute

November 20, 2023

Abstract

Software development requires developers to juggle and balance many information-seeking and understanding tasks. From determining how a bug was introduced, to choosing what API method to use to resolve the bug, to how to properly integrate this change, even the smallest implementation tasks can lead to many questions. These questions may range from hard-to-answer questions about the rationale behind the original code to common questions such as how to use an API. Once this challenging sensemaking is done, this rich thought history is often lost given the high cost of externalizing these details, despite potentially being useful to future developers.

In this thesis, I explore different systems and methods for authoring and using this rich *meta-information*. Specifically, I have developed systems for *annotating* to support developers' natural sensemaking when understanding information-dense sources such as software documentation and source code. I then demonstrated how this meta-information can be harnessed and used in new ways, including for assessing the trustworthiness of documentation and for capturing design rationale and provenance data of code.

To begin exploring to what extent meta-information for software developers may be utilized to support sensemaking during software development tasks, I developed Adamite, a browser extension that enables developers to annotate and organize their questions, open tasks, issues, and other thoughts about API documentation. Adamite was inspired by the fact that developers often face barriers brought upon by questions and issues with API documentation and the insight that it is likely that other developers have already experienced those barriers and eventually overcame them. Indeed, when using developer-annotated documentation, participants were able to complete significantly more of a challenging programming task compared to the baseline. Given this success, I generalized Adamite's annotating approach to the IDE with the Catseye plugin for Visual Studio Code, and focused on helping the original developer better keep track of information to answer their own questions. The following project, Sodalite, expanded the Catseye approach to support long-form content through leveraging the relationship between the source document and annotation anchor points to serve as a signal for the "health" of the document. To lower the information authoring cost and account for the issue of scale, I then created the Meta-Manager which automatically versions and extracts meta-information about code and its provenance. Through all of these projects, this thesis has explored the ways in which meta-information may be presented and how the unique properties of code and the rich contextualized information about it may be harnessed to help developers in supporting their information seeking and understanding tasks.

While these systems have worked well in isolation, each only tackle a subset of the types of information developers need and utilize only some of the available meta-information about code. Further, given the shifting nature of software engineering tasks with the rise of AI code-generation systems, we have shown that our system can track and collect some of this information. To conclude this thesis, I propose extending the Meta-Manager to work with my other systems to explore to what extent combining these forms of meta information can answer otherwise unanswerable questions, while extending the systems to capture other significant types of meta information, such as CoPilot prompts and GitHub commit messages. Similar to my previous work, I plan to evaluate the integrated system through experiments to understand to what extent these new classes of meta-information can actually help developers answer their real questions about code.

Contents

Chapter 1

Introduction

Developing software requires developers to keep track of many types of information while performing various interleaved tasks. For example, to debug some code, a developer must first navigate through the code to determine what part or parts of the code are responsible for the bug [85, 92, 128], run the code and diagnose the output to understand how and when the code fails [87, 92, 128], write code and possibly research solutions online to fix the bug [20, 94, 170], then test the code again to ensure the solution worked [48]. Typically, the developer's mental model of this rapidly-evolving problem space is not externalized [24, 98, 124], which can be problematic as working memory is limited [62, 105, 157]. Further, development processes are often iterative [82], can be interrupted [123, 124], and may just be one task amongst many that the developer is handling [83], leading to further cognitive strain.

Given this significant cognitive cost, many research projects have enumerated the challenges in both writing and understanding code. Just some of the challenges include maintaining task awareness [104, 123, 124], making sense of external libraries and their documentation [4, 65, 115] (with software documentation, itself, having its own set of well-known problems [4, 93, 118, 137, 163]), understanding the rationale behind the current code [83, 91, 104, 146], determining what part or parts of the code are relevant to the change the developer is introducing [82, 88], and designing that change and implementing it [88, 94]. Completing these smaller information-intensive tasks often results in the creation of knowledge that is lost, either because it is not externalized by the progenitor of the information or because it is not logged.

Some qualities of programming make tracking this information especially challenging in comparison to other complex sensemaking domains. For one, a developer's end-state is not often reaching an answer or decision to these implicit questions, but is to utilize that information to complete some programming-related task, such as fixing a bug. Secondly, the information landscape is often changing as the developer introduces new code, which can produce new runtime behaviors and introduce new questions. This means that a) the developer does not normally have the mental bandwidth to externalize this information beyond, perhaps, a short note [68, 98]; b) this information is implicitly associated with the corresponding code that stems from the gained knowledge [68, 94]; and c) as the code changes, the information the developer learns is highly contextualized to the time in which it was discovered or created [13]. We call this information created as a byproduct of authoring or making sense of developer materials (e.g., code, web resources, documentation, etc.) *meta-information*.

In software development, one of the most ubiquitous and well-known meta-information types are code comments. Often created as a way of keeping track of bugs [155] and open to-do items [153, 154], documenting code [142], or informal versioning of code [75], code comments can serve as rich information about what a developer is doing when managed with tooling [161]. Other forms of meta-information about code include Git commit messages, code documentation, code versions, and so on. The act of developing software produces a vast and dense information landscape comprised of these various forms of meta-information. Meta-information also exists in other information authoring domains such as Microsoft Word or Google Docs comments.

In this thesis, I[1] explore developing software systems to capture and present programming-related meta-information to assist in overcoming known challenges in developing software. This work is predicated on the knowledge that many forms of meta-information are generated at different times during the software development life cycle and this knowledge, when captured and presented in a comprehensible manner that leverages the code or other source information within the developer's working context, can be useful. By associating this meta-information with code, the connection between what the developers care about (i.e., the code) and all of this other information that normally is not captured but can actually answer developers' implicit questions about code (e.g., visited web pages for where some code came from, code edit history for when some change occurred, added and removed and commented out code for reasoning about what a developer has tried, etc.) is an effective way to keep the developers within their working context, while providing them the information they need to answer their questions.

The research efforts investigate the claim that meta-information, if properly contextualized given its relationship to some code or documentation, can be both useful to the initial developer and later readers. For the initial author, externalizing their thoughts and questions as meta-information can help with keeping track of information while completing cognitively-demanding software engineering tasks. For later readers, these traces of the context in which the initial development occurred can help with answering otherwise unanswerable questions about code and for better understanding unfamiliar code and learning resources. I also show that code meta-information can be used, not only for reasoning about some code, but also for evaluating written materials about code, given the connection between meta-information and its corresponding code. An overarching goal of this thesis is to treat information about code as a first-class entity, not unlike the code itself, given that the rich history of code and its development is often the *answer* to developers' questions, but is not typically captured in a systematic or easily explorable manner.

1.1 Overview

There exists a vast space of meta-information about code, with varying levels of current tooling support. For example, both large commercial projects and smaller-scale academic studies have extensively explored code versioning, one form of code meta-information, including enterprise-level products like GitHub [109] and research projects such as Variolite [75] or Chronicler [168]. Other areas of code meta-information have been less explored, including developers' notes written about code, despite prior research confirming that this is an activity that developers sometimes perform [98, 102, 104].

[1]The projects discussed in this thesis were lead by myself but were developed in collaboration with other researchers – out of respect for their contributions, I will predominately use "we" when discussing the various projects and papers that comprise this thesis.

To begin exploring to what extent meta-information can be used to help software developers overcome their sensemaking barriers, I began by exploring *annotations* as a vehicle for associating meta-information with some source information, thus establishing a shared context between author and later reader. To this end, I designed the Adamite system, a web browser-based annotation tool specifically designed for annotating software documentation, with features designed to combat known documentation usage barriers. We chose to focus on documentation, given the large body of software engineering research discussing particular pain points that we expected meta-level notes to address. For example, some prior literature has discussed the challenges in finding pertinent information in documentation and how this information may be fragmented across multiple locations within the documentation [4, 37, 163] – to address this barrier, Adamite allows developers to add multiple text anchor points to their annotation, such that all relevant parts of the documentation may be connected to one note. Adamite demonstrated the efficacy of utilizing developer-authored meta-information presented as annotations to assist later developers in their own programming tasks – developers using Adamite annotations performed significantly better on a programming task.

Adamite was successful in helping later developers utilize meta-information about code to overcome their information barriers – however, we found that the initial authors did not experience a significant effect, despite appreciating the annotation tool. Given the need for more support for the initial author in utilizing their meta-information and the increased need for information tracking support in the IDE, I developed Catseye, a Visual Studio Code [110] extension for annotating code. Catseye not only supported adding free-form text to code that was abstracted away from the original source code (as opposed to code comments) but also supported other developer information-tracking activities including collecting output data and lightweight versioning of code, with the option to annotate these other forms of meta-information. Developers using Catseye performed better when attempting to keep track of information while debugging some purposefully-confusing code.

In active usage of Catseye, it became clear that the extremely mutable nature of code can lead to information within the annotations quickly becoming out-of-date. While Catseye preserves additional meta-information about when the annotation was created, edited, and what Git version history is relevant to it, if the code changes such that the annotation content is no longer relevant, there is no way to capture that. Considering out-of-dateness is a large problem in documentation as well, the insight that the connection between code and text is both always changing and the accuracy of the link can be used as a signal for the trustworthiness of the document led to us extending Catseye for long-form documentation with the system Sodalite. Long-form documentation can be a useful information source, given that it co-evolves with the code, but practice shows that does not always happen[93]. Sodalite uses the connection between code and text and its ability to connect the code to the text as a metric for how "healthy" the document is in the user's current programming context. This system showcases how meta-information can be used for new activities beyond simply displaying data.

Adamite, Catseye, and Sodalite were all successful in supporting developers in authoring their own meta-information, but, in their designs, were reliant upon the programmer to actually write the information. Further, what developers have to say about code is only one type of meta-information and is incapable of answering some of the types of questions developers may have, such as *when* a particular change was introduced. To expand the types of questions we could support answering with meta-information and to lower the cost of authoring useful information, we developed the Meta-Manager, a Visual Studio Code extension with a supplementary Google Chrome extension, that automatically listens for and captures editing events of interest in both the editor and the browser. Some editing events of interest occur *within* the editor including copy-pastes between code patches, which may inform where some code came from and when, and code commenting, which may show the author trying different solutions or the author adding or changing documentation. Other edits of interest occur in the *web browser* – we are particularly interested in copy-pastes from the browser (e.g., code examples taken from official documentation, ChatGPT code snippets, or answers from Stack Overflow posts) into the editor, which may inform why some code is written the way it is, given relevant Google Chrome searches and/or visited web page(s). AI-generated code from ChatGPT, in particular, contains a rich collection of interesting additional meta-information about the development of the code, such as the initial query and chat title, which may inform what the developer's original intent was. Indeed, in our study developers were able to use the Meta-Manager to answer otherwise unanswerable questions about code using this rich version history, with insights about AI-generated code presented as code meta-information being particularly valued by participants.

While the meta-information I have built systems to capture, organize, and use has been useful for helping developers both keep track of their own information and answer questions about other developers' work, I believe this information will only become more important as software engineering tasks move further away from simply writing code and towards usage of AI-generated code. I envision future software engineers will spend more of their time designing and prompting AI tools to write code, then integrating and reviewing this code. All of these activities are likely to produce new meta information that can answer new classes of questions developers may have about this AI-generated code. For example, a developer may wonder whether or not the code is AI-generated, what the original prompt was to produce the code, how the code and prompt evolved over time, what has happened since this code's introduction, and so on — all in service of understanding the rationale behind the code and how their planned contributions will need to be designed in order to work in harmony with this code while retaining the original intent. I propose extending the Meta-Manager and my other systems to work in this new paradigm through integrating the systems to capture even more of developers' generated code meta-information.

1.2 Expected Contributions

The series of work discussed in this thesis explores the utility of supporting developers in both authoring and utilizing meta-information to better make sense of code and documentation. Specifically, the systems provide platforms for authoring and sharing useful and contextualized information about code (Adamite, Catseye, Sodalite), unify various forms of meta-information into one platform with a single interaction method (Catseye, Meta-Manager), and demonstrate how meta-information may be used to overcome known barriers in documentation usage and answer longstanding, hard-to-answer questions about code (Adamite, Sodalite, Meta-Manager).

Thus far, my work has made the following contributions:

- A review of literature around developer information needs, systems developed to support some of these needs, and how meta-information about code has or can be used to combat these challenges (chapter 2).

- Identifying that short, easy-to-author notes, when attached to programming-related documents including code and documentation, can address known challenges in information tracking and usage (chapters 3 and 4).
- Adamite, an annotation system designed with specific features to use meta-information as annotations on documentation to overcome known barriers in using software documentation [65] (chapter 3).
- Evidence that developers' notes can be useful for other developers when properly presented given the implicit context communicated through annotated text (chapter 3).
- Catseye, a code annotation system, with features to help a developer keep track of code-related meta-information using a unified annotating interaction [68] (chapter 4).
- A documentation authoring and maintenance system, Sodalite, that utilizes code meta-information to support developers in assessing the validity of documentation and finding appropriate parts of the code to document [67] (chapter 5).
- A system, Meta-Manager, that can collect, index and store a larger collection of code editing and provenance meta-information that has never previously been collected at scale for answering otherwise unanswerable questions about code history [66] (chapter 6).
- A series of lab studies showcasing the usability and utility of these tools in supporting developers' varied information needs in different contexts.

To complete this thesis, I plan to make the following contributions:

- A prototype system that incorporates and unifies all of the types of meta-information the prior systems supported, while supporting new forms of meta-information, to assist developers in answering a new class of questions about code.
- An evaluation of the prototype system that assesses its ability in supporting developers in their sensemaking of code.

Chapter 2

Background and Related Work

Developers are tasked with managing many different types of information when both authoring and making sense of code. In the following sections, I discuss prior literature about how developers make sense of code, tooling support for that process, meta-information about code (including documentation), and prior systems to support the creation and management of some of these types of meta-information.

2.1 Making Sense of Code

Researchers in both human-computer interaction (HCI) and Software Engineering have extensively studied how developers make sense of or comprehend code [18, 20, 40, 43, 83, 85, 86, 87, 89, 104, 138, 139, 144]. These studies are typically situated in the context of understanding unfamiliar code [40, 83, 89, 104, 139], which is a common and significant challenge for developers [115]. For example, developers need to understand unfamiliar code when learning a new API [39, 42, 43, 69, 98, 115, 116, 137], joining a new code base [14, 36, 74, 135, 151, 162], adapting code from online sources [10, 20, 98, 117], and so on. Some of the challenges developers must overcome when understanding unfamiliar code include maintaining their task awareness [102, 112, 123, 124, 154]; developing mental model of the code [18]; questions, hypotheses, and facts learned about the code [51, 91, 146]; the codes' current version and output [75, 92]; and "working set" of code patches [18]. Even when an implementation task is small or the code is more familiar, developers must continually keep track of these varied information types for maintaining software to varying degrees of success [82].

2.1.1 Developer Tools

Given the diversity of types of information developers must keep track of, a number of research tools have focused on supporting these different activities. One challenge developers experience when making sense of code is navigating through code and finding and relating code patches of interest (sometimes referred to as the "working set") [18, 32]. Researchers have investigated various solutions to ease navigation costs including clustering code patches as "bubbles" [18] or in a dedicated workspace [1, 32] and augmenting code comments to serve as navigational way points [54, 153, 154, 161]. Other projects have attempted to ease navigation through suggesting code patches, such as to assist in fault localization [48, 127, 128] and for helping newcomers to code projects find where to begin [13, 36]. In all of these cases, the code patches have a higher-level semantic meaning as to why they are relevant to the developer – a relationship which is only sometimes made clear given the tool. All of my tools attempt to make the relationship between this meta-information and its corresponding code traceable and bi-directional to assist in comprehension and navigation.

Other projects have focused on leveraging the large amount of already-written code and developer resources that exist online to assist in overcoming cognitive barriers incurred when programming. These projects commonly suggest code snippets taken from documentation [19, 107, 120], open source repositories [159], or question-answer forums [71, 173], given a developer's query. The underlying rationale for this design, namely that developers care about code first and foremost, also drives my design approaches, but my systems capture other forms of information beyond just code since code, alone, cannot answer every type of question. Other systems lower the barrier of foraging for programming-related information on the web through integrating web-based functionalities (e.g., web search) into the IDE [34, 56]. My systems, especially the Meta-Manager, attempt to also better connect the activities that are naturally happening both in the editor and the web browser, but do not require the user to adopt a new browsing or development environment.

2.2 Meta-Information About Code

2.2.1 Writing About Code

One strategy developers employ when keeping track of information is writing down, either through code comments or external notes, what they want to keep track of. In the case of code comments, research has explored what types of information are in code comments [142, 153, 155, 167], whether the code comments are actually useful [17, 131, 167], and how these comments are later used [49, 132] and cleaned up [155, 156]. An analysis of 2,000 GitHub projects found 12 distinct categories of comments developers write and found that information designed for code authors and users appeared less often than more formal types of documentation, suggesting that developers are not frequently commenting for their own benefit or these comments are removed prior to submitting the code to publicly-viewable repositories [142]. Other meta-information about code that is sometimes included in code comments are links to where some code originated from, in the case that the code was copy-pasted from online [9, 57].

Developers also employ note-taking for tracking information about code [33, 98, 102, 104, 123]. These notes are commonly used as memory-aids [33, 98, 123] both for themselves and for communicating insights to other teammates [29, 104], for keeping track of useful resources [98], and for keeping track of their progress on a development task [102, 123]. Notably, these notes often lose their initial utility given their lack of context, while others are authored with the full expectation that they will be thrown away [98].

Code development and coordination with a software team commonly results in the creation of other types of written works. This includes version control system (VCS) commit and pull request messages [96, 101, 160], computational notebooks [58, 76, 129, 165, 166], bug reports [16, 133], and code review messages [8, 53, 90, 113, 114, 140]. Developers may also write about their code with the intent of helping others, whether that be a blog post [121, 122, 125], question-and-answer forum post [7, 169, 175], or tutorial [58, 60, 125, 158, 164]. Most of these written artifacts suffer from the same challenges that other written works about code face in that they are typically abstracted away from the original code context, thus making them difficult to discover or glean answers from.

Documentation

Perhaps the most studied type of writing about code is software documentation. Much prior literature has investigated what information is written in documentation [61, 103], what the problems are with that information [3, 4, 26, 45, 93, 108, 118, 137, 139, 163], how this documentation is authored [38, 143, 152], and automating documentation processes to offset authoring costs and standardize the information present in documentation [2, 55]. Notably, the majority of this documentation work is in the context of software documentation for end users of a software library, e.g. API documentation [115]. Slightly less work has focused on documentation created for other developers working on the same code base, such as internal documentation about the code base [135, 143] or open source on-boarding files [158], where the primary goal is to help other developers understand and contribute to the code base.

Once some software documentation is made, researchers have studied how developers maintain those documents and use that information. In studies of usage, researchers have identified many problems of documentation that lead to the documentation being less trustworthy [100], including questions about how up-to-date the information is [4, 93, 163] and how complete the information is [137, 139]. Maintaining documentation has also been found to be challenging, with developers reporting on the costly time spent on updating documentation [163] and a lack of clarity on where and how to update the documentation given code changes [50].

2.2.2 Other Meta-Information

Developers keep track of many other types of information that are not or cannot be written down. One ubiquitous form of meta-information about code is code history, i.e., code versions across time. Most software engineering utilizes a version control system (VCS) (e.g., Git) for keeping track of code releases and to make collaboration across potentially many different versions of code possible. However, in between formal check-ins of code, developers have reported a need for keeping track of their code versions. Whether that be in the context of data science programming where intermittent versions may represent small-scale experiments [59, 75, 76] or when maintaining a code project and trying different solutions for fixing a bug [172] or adapting a code example [20], developers employ a variety of methods for keeping track of these intermittent versions such as commenting out the code [75]. Both these less formal intermittent code versions and committed code versions can serve as meta-information about the "current" version of the code in service of, e.g., understanding how some code evolved over time.

Another form of code meta-information closely related to code versions are outputs generated by code. Outputs may be as simple as a console log message or as complex as a full graphical user interface. In either case, developers must often keep track of how these outputs change with respect to their code edits [75]. This may happen when the developer must revert to a prior, functional version after a bug is introduced or to compare differences in outputs with the intention of choosing an optimal version [172]. Some tools have been developed to keep track of outputs with respect to the code versions that generated them [76], while others have focused on making the output itself more useful through improved interaction and flexibility [73].

In our work, we design tooling approaches to support the authoring and management of programming-related meta-information for sense making. Whether this meta-information is in service of externalizing and tracking some thought about code, an intermittent code version, a piece of useful documentation, or the history of a function, the proposed work will unify all of these forms of normally-siloed and de-contextualized information pieces into a singular system.

Chapter 3

Adamite: Meta-Information as Annotations on Documentation

This chapter is adapted from my paper:

[65] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. "Understanding How Programmers Can Use Annotations on Documentation". In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA*. ACM, New York, NY, USA, 16 pages.

3.1 Overview

Application programming interfaces (APIs), including libraries, frameworks, toolkits, and software development kits (SDKs), are used by virtually all code [115]. Programmers at all levels must continually learn and use new APIs in order to complete any project of significant size or complexity [42]. In learning APIs, developers depend upon the documentation, including tutorials, reference documentation, and code examples, along with question-and-answer sites like Stack Overflow [93]. However, developer documentation is known to be problematic with common and significant issues including incompleteness of information, out-of-date information, and fragmented information [3, 4, 137, 163].

In this chapter, we investigate how developer-authored meta-information about documentation, presented as annotations, may help developers overcome some of these barriers. We hypothesized that the highly-contextualized nature of developers notes when anchored to text within the documentation, can help the initial annotator better utilize their notes about documentation. Further, when given proper tooling support, this information can be leveraged by later users of the documentation to overcome their documentation barriers.

To explore the concept of annotations as a way of supporting short notes on documentation that are useful both for the author and for later readers, we started with a preliminary lab study that explored the concept of annotations on documentation using an off-the-shelf Chrome extension, Hypothesis [72], and then we performed a corpus analysis of annotations on documentation created using Hypothesis. Given what we learned from these preliminary analyses, we developed our own documentation-specific annotation tool, Adamite[1]. Next, we ran a two-pass user study where we explored the kinds of annotations developers *authored* when learning a new API and then had another set of developers *read* those annotations while attempting to complete the same API learning task using Adamite. We compared these participants to a *control* condition which had no annotations. From these studies, we provide evidence that annotations are useful in helping developers overcome documentation-related issues.

3.2 Preliminary Studies and Design Goals

3.2.1 Lab Study with Hypothesis

To explore the efficacy of annotations as a useful learning device for API learning tasks using documentation, we ran a preliminary study where people learned an unfamiliar API while using Hypothesis [72]. In summary, we found that developers are able to use annotations in the ways we envisioned, but that annotation authoring and reading could be improved for developers by adding additional tooling features.

Study Design

The preliminary study had two distinct phases: the first phase was focused on understanding how developers *author* annotations during an API learning task, while the second phase focused on how developers *read* annotations that are already attached to documentation. In each phase, the 4 participants completed an API learning task adapted from my previous study [64] that required them to forage through API documentation to complete a programming task.

In the *authoring* condition, participants were given the documentation with no annotations, and instructed to add annotations when they learned anything useful, had questions about the content in the documentation, or had any other thoughts about the documentation. In the *reading* condition, participants were given the same documentation, but with annotations added. The annotations included annotations authored by the first author that were designed to be helpful for this task given what developers were confused about in a previous study [64], and other annotations that were designed to be "distracting" to simulate the more realistic case where not all annotations would be relevant. In total, we had 23 "helpful" annotations and 44 "distractor" annotations, totalling 67 unique annotations.

Results

In the authoring condition, the 4 participants together authored a total of 19 unique annotations. Each participant, on average, authored 4.75 annotations, with annotations averaging 4.41 words. Hypothesis also allows users to simply highlight a piece of text on a web page without adding any text content to the anchor (hereafter referred to as "highlight" annotations) — out of the 19 unique annotations authored, 5 were these simple highlight annotations.

Many of the annotations that participants created showed a part of the documentation that illustrated how to achieve some part of their current task. Other

[1]Adamite stands for **A**nnotated **D**ocumentation **A**llows for **M**ore **I**nformation **T**ransfer across **E**ngineers and is a green mineral.

annotations served as task reminders or open questions the author had about the documentation content.

In our analyses, we also found that participants had many questions about the documentation (on average, 10.8 questions per participant) which were not annotated. While trying to answer these questions, participants routinely encountered more confusing information, resulting in them losing track of their original questions. Given this confusion, participants struggled to answer their questions, with only 29% of questions definitely answered. Notably, Hypothesis does not have any way of marking or following up on a question.

From this initial preliminary study, we found evidence that annotations may enhance the original text. Additionally, we found support for different types of information needs that are not directly supported by Hypothesis, such as keeping track of open questions. In the reading condition, we also learned through our "distractor" annotations that annotations need to be easy to skim and relatively short in length and more support for discovering annotations, either through more anchor text points or filter and search.

3.2.2 Corpus Analysis of Hypothesis Annotations

In order to supplement our preliminary study, we queried Hypothesis's API to get a list of public annotations which developers have already made on official API documentation including public APIs from Google, Microsoft, Oracle, and Mozilla, along with other developer learning resources including Stack Overflow, W3Schools, and GitHub.

Across these sites, we found 1,995 public annotations. Of the 1,995 annotations, 196 were questions about the content of the documentation, and 995 were highlight type annotations. Of the 1,000 annotations with content, 16 of the annotations were to-do items the author wanted to follow up on, 43 pointed out problematic aspects of and potential improvements to the documentation, and 79 were created to specifically call out important or useful parts of the documentation[2]. These annotations were authored by 298 unique users (average = 6.694 annotations per user, minimum = 1, maximum = 677) across 1,143 unique web pages. The authored annotations were, on average, 8.79 words long and were anchored to text that averaged 12.35 words.

We believe some of the 1,000 annotations that contained content could benefit other developers with additional tooling support. For example, a Hypothesis user annotated the text "you can pass the path to the serve account key in code" and asked how they can do that. This user then later annotated a code example showing how to achieve this behavior at a different point in the documentation and said "finally found it". While these two annotations depend upon one another in order to make sense and point to different parts of the documentation, Hypothesis does not allow for these annotations to reference one another, suggesting a need for better tooling support for multiple anchors for annotations and keeping track of open questions.

These annotations provide support for our claim that some developers are willing to write annotations and attach them to documentation, as they are already doing this. Moreover, the annotations we found follow some of the patterns we identified in our preliminary study, such as open questions and issues. However,

[2]These counts were generated by counting instances of phrases like "incorrect" and "todo" in the annotation content, then manually reviewing all of the annotations that contained those phrases to determine if they were actually e.g., an issue or todo item.

Hypothesis's general-purpose annotation system does not have enough support to effectively utilize these annotations.

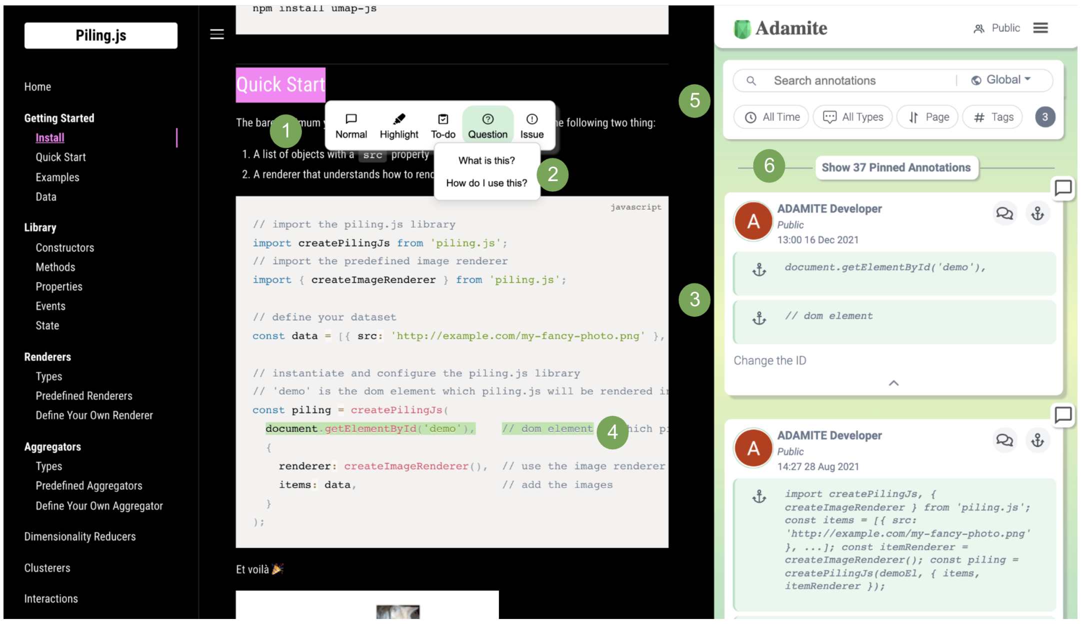
3.2.3 Design Goals

Given prior literature and what we discovered in our preliminary studies, we developed the following design goals for our system:

- **Support developer note taking through annotations.** Developers sometimes take notes on what they have learned [98, 102, 104, 123], even when using documentation (Section 3.2.2).
- **Support developers' question-asking and answering.** Developers have many questions about unfamiliar APIs and their documentation ([42, 146], Sections 3.2.1 and 3.2.2).
- **Support diagnosing documentation issues.** Developers commonly identify documentation issues including obsoleteness [4, 163], incorrectness ([4, 163], Section 3.2.2), incompleteness [4, 27, 37, 136, 137, 163, 176] and ambiguities [4, 136, 137, 163] but have no way of sharing that information.
- **Support developer task-tracking.** Developers take notes on open tasks that they must work on, especially when interrupted [123], and occasionally take notes on tasks they must complete that are related to parts of the documentation they are reading (Section 3.2.2).
- **Support connecting related parts of the documentation.** Developers need to build up a mental representation of an API [69, 80, 98] and connect related resources [4, 37, 163].
- **Support better discovery of important parts of the documentation.** Developers, especially selective [21] and opportunistic [20] learners, want to quickly find information that is relevant to them ([108], Section 3.2.1).

3.3 Overview of Adamite

We designed Adamite, a browser extension, specifically to help developers keep track of important information, organize their learning, and share their insights with one another. To create an annotation, a developer, who we call the annotation "author" simply needs to open the Adamite sidebar, highlight some text on the web page (called the "anchor"), select the type of annotation, optionally add text to a rich text editor that appears in the sidebar, and click on the publish button. Once published, the text that the user annotated will be highlighted on the web page and the annotation will appear in the Adamite sidebar – see Figure 3.1. Users may also add tags and additional anchors to the annotation. Annotations can be published publicly, privately, or to a group of Adamite users. Once an annotation has been published, it may be replied to by others, and edited or deleted by the original author. Clicking on the anchor icon on the annotation will scroll to the part of the web page the annotation is anchored to (or will open a new tab if the anchor is on a different page) – conversely, clicking on the highlighted text on the web page will scroll to the corresponding annotation in the sidebar.



```
npm install umap-js
```

The bar mum ... e following two thing:

1. A list of objects with a `src` property
2. A renderer that understands how to rend

```javascript
// import the piling.js library
import createPilingJs from 'piling.js';
// import the predefined image renderer
import { createImageRenderer } from 'piling.js';

// define your dataset
const data = [{ src: 'http://example.com/my-fancy-photo.png' },

// instantiate and configure the piling.js library
// 'demo' is the dom element which piling.js will be rendered i
const piling = createPilingJs(
  document.getElementById('demo'),    // dom element  hich p
  {
    renderer: createImageRenderer(),  // use the image renderer
    items: data,                      // add the images
  }
);
```

Et voilà 🎉

```
document.getElementById('demo'),
```

```
// dom element
```

Change the ID

```
import createPilingJs, {
createImageRenderer } from 'piling.js';
const items = [{ src:
'http://example.com/my-fancy-photo.png'
}, ...]; const itemRenderer =
createImageRenderer(); const piling =
createPilingJs(demoEl, { items,
itemRenderer });
```

FIGURE 3.1: Adamite's sidebar (on the right) open on an already-annotated web page in the browser. (1) shows the pop-up for when a user selects some text – at this point they can begin creating a new annotation by selecting an annotation type. (2) shows the menu of question annotation prompts users can choose from. (3) shows a published normal annotation with two anchors. (4) shows how the annotated text appears on the web page. (5) shows Adamite's search and filter pane. (6) shows the pinned annotation list button.

One goal of Adamite is helping developers structure and share what they learn in the documentation in a way that is useful both to themselves and for later developers. To achieve this, we developed annotation types. In addition to the typical "normal" (with a user-written comment) and "highlight" (just the anchor and no comment) annotations, Adamite supports question, issue, and to-do annotations. We chose these three annotation types to assist developers in keeping track of their questions, to point out and possibly attempt to rectify issues found in the documentation, and to help them keep track of their tasks. Issue annotations have a button intended to alert key stakeholders, such as the documentation writers, of the described problem with the documentation. Question annotations are stateful, meaning unanswered annotations will stay available until the developer either marks the question as "answered" (at which point the answer will be appended to the original question), or marks the question as "no longer relevant". To-do annotations are also always available until they are marked as complete.

Question and to-do annotations are always available using Adamite's "pinning" mechanism. Most annotation systems only show annotations that are on the user's current web page. However, considering that documentation may be spread across many pages and developers may visit many web pages when attempting to complete a programming task, we added in the ability to *pin* an annotation, such that it is always available in a list at the top of the sidebar. To-do and question annotations are pinned by default, since the developer is unlikely to find their answer or finish their task while they are on the same web page.

Given that a common documentation problem is fragmented information, we found a need to support *multiple* anchors for a single annotation. This feature can be used to connect parts of the documentation that the user feels should be presented together, or to better contextualize their annotation. Developers may also use anchors as a way of collecting multiple parts of the documentation that they feel are related to one another given the developer's task and their evolving understanding of the API.

For later users of the annotated documentation (who we call annotation "readers," but can be the same person as the annotation authors), it is likely that not all of the annotations are relevant to what the developer is trying to do. To help readers find the most relevant annotations, we support search (using Elasticsearch [44]) and filters (see Figure 3.1-5). Readers can search across a web page, website, or across all of Adamite's annotations and filter on the annotation type, when the annotation was created, and what tags the author has tagged the annotation with. Readers may also sort the annotations by their location on the page or by the time at which the annotation was authored.

3.4 Lab Study

In order to understand the role that annotations play in developers' documentation usage while learning a new API and using Adamite, we ran a lab study with three conditions to understand how developers create and use annotations. Participants in one condition *authored* annotations while completing an API learning task, and participants in the second condition *read* these participant-authored annotations. The third condition was a *control* condition where participants completed the same API learning task using just the documentation. The lab study consisted of a training task, a programming task, and a survey to assess the participant's background.

3.4.1 Method

Training

Each condition included a training exercise using Tippy, a React library for making tooltips, and its documentation to either familiarize the participants with Adamite and its functionality (Adamite conditions) or to familiarize them with thinking aloud while reading through documentation (control). Participants in the Adamite conditions learned how to create an annotation of each type, reply to an annotation, add an additional anchor to an existing annotation, search, filter, edit and delete an annotation and practiced thinking aloud while performing these tasks. The control condition practiced thinking aloud when they had a question, found an answer to their question, and identified an issue in the documentation.

Task

For the task, participants were asked to complete an image aggregation and organization task using Piling.js (hereafter referred to as "Piling"), a JavaScript library for handling visualizations [126]. Piling was chosen as it is a relatively small library, meaning the participants would have adequate time to gain a high-level understanding of the library during a lab study.

The task was to use Piling to take a set of four provided images and render and sort the images (see Figure 3.2 for the output and detailed steps). The task was chosen as, despite its apparent simplicity, it requires the participant to learn some of Piling's core concepts. Participants were objectively graded upon how many of the 4 steps they were able to complete correctly. To start, participants were given a JavaScript file containing comments stating the goal of each step.

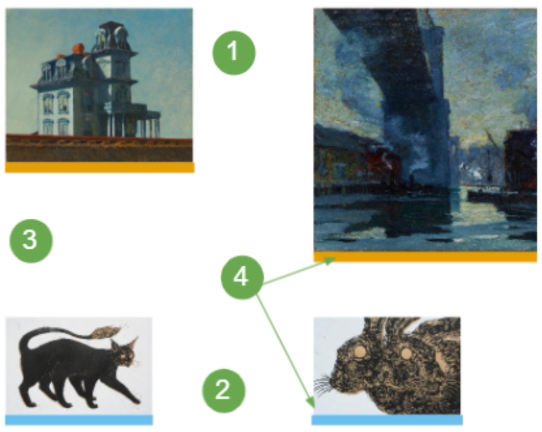


FIGURE 3.2: The correct output for the task. Each number refers to the step number. (1) creates and renders the 4 images. (2) puts the images in 2 rows. (3) arranges images by a user-defined property using the `arrangeBy` method. (4) requires the user to set a label on their data, such that elements with the same label will have a matching stripe along the bottom of the picture.

In addition to completing the programming task, participants were asked to think aloud and to pretend as though they were in a small team learning Piling. Dependent upon the condition, further instructions differed slightly. *Control* condition participants were told that they needed to relay what they had learned to their teammates in whatever way they would normally do so, such as note taking. Adamite *authoring* participants were instructed to create annotations with any questions or thoughts they had about the documentation, issues they found in the documentation, and thoughts they wanted to follow up on and that these annotations would be shared with their future teammates. Each authoring participant started with an un-annotated version of the documentation. In the Adamite *reading* condition, participants were given annotated documentation and were told to pretend that the annotations were created by a teammate who had already learned Piling and were instructed to speak aloud when an annotation was helpful or unhelpful. Participants in the reading condition were not required (but were allowed) to create annotations or interact with the annotations present in the documentation.

Annotation Selection

In choosing annotations to include in the reading condition, two researchers separately coded each annotation created during the authoring condition for whether or not to include it. Inclusion criteria included identifying matching annotations across participants to select which one was the clearest, most appropriately anchored, and concise – qualities informed by our preliminary study and others [5]. The predominant cause for the majority of annotations to be removed was redundancy – participants commonly annotated information related to the first two steps of the task (see Figure 3.3)[3]. We also excluded annotations that the participant later stated were incorrect or that the participant deleted, and annotations that lacked sufficient context (including all highlight annotations). Finally, we omitted to-do annotations, as they are designed only for the original author's usage. The researchers had a 71% agreement – in the cases where the researchers disagreed, they had a discussion until agreement was reached. Through this process, we were left with 31 annotations.

[3]Note that this is due to being a lab study – in a realistic situation, people would likely read an existing annotation and not create a redundant one.

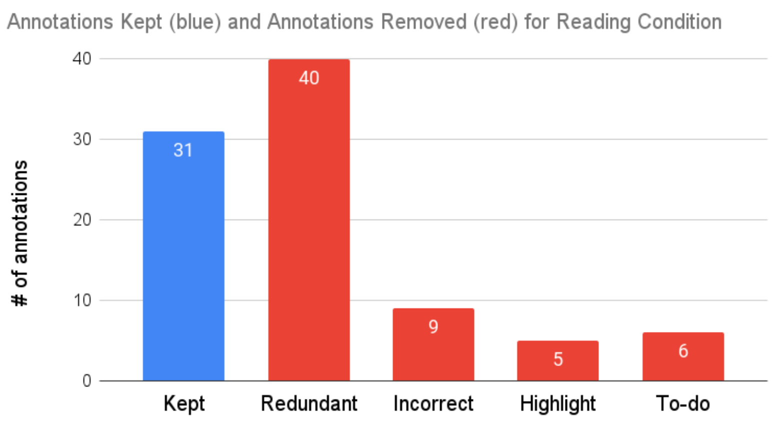


FIGURE 3.3: The number of annotations removed for each reason, along with the annotations kept, out of the 91 total annotations. 2 highlight annotations were retained as the users edited them to add text, making them semantically identical to normal annotations.

One additional annotation was added by the first author to assist with the final step of the task, as no participant in either the authoring condition or control condition got to that point of the task. After this process, we had 32 annotations, with 18 normal type annotations, 10 issue type annotations, and 4 question annotations, 3 of which were answered[4]. We did *not* omit any annotations due to relevance or correctness, since we wanted to leave in anything that at least one participant wanted to comment on to be more realistic.

Participants

We recruited 31 participants using departmental mailing lists at our university and social media. One participant could not finish the study due to technical difficulties, so we only report on the 30 who completed the whole study. Each condition included 10 participants and were randomly assigned between the authoring and control condition – the reading condition occurred after the other two conditions so all remaining participants who signed up were assigned to that condition.

All of the participants were required to have some amount of experience using JavaScript, not to have used Piling before, and to have been programming for at least 1 year (actual minimum: 1 year, maximum: 20 years, average: 7.98 years). The participants' professions included graduate students in computer science-related fields, user experience researchers with a computer science background, and professional programmers. The gender breakdown of our study consisted of 19 men, 9 women, and 1 non-binary person. Participants across each condition had a similar amount of JavaScript experience and years of programming experience.

All study sessions were completed remotely using video conferencing software. Participants were audio and video recorded, and each participant's session took approximately 90 minutes, with 45 minutes of that time allotted for the programming

[4]We included one unanswered question to account for the realistic situation that not all questions would be answered and because the question asked was a common question among participants – notably, answering this question was not necessary for succeeding in the task.

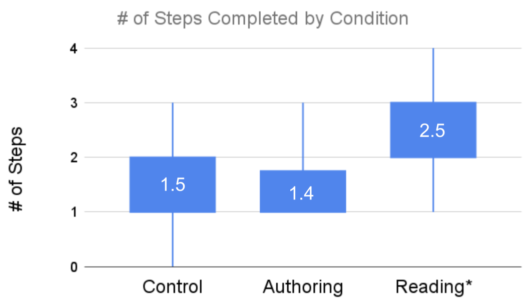


FIGURE 3.4: The difference between reading and each of the other two conditions is statistically significant, but the difference between control and authoring is not. The average number of steps completed is in the center of each box.

task. Each participant was given access to the Piling documentation and a CodeSandbox.io [22] project which had JavaScipt, HTML, and CSS files with Piling installed and a photo of the output, along with written-out steps for the task. Participants were compensated $25 for their time, save for 2 participants who elected not to be compensated.

Analysis Methods

Across all of the conditions, we objectively graded participants on whether or not they succeeded in completing each of the 4 steps outlined in the task instructions. In the Adamite conditions, we analyzed the video recordings and log data to count how many annotations participants authored, and how often they interacted with Adamite and its annotations.

We qualitatively coded the annotations developers made in order to characterize developers' annotating strategies. Using an open coding method, two authors coded the normal type annotations by independently coding each annotation and refining categories based upon their individual codes. For issue and question type annotations, we coded the annotations dependent upon what issue in a list of commonly defined issues [4] was identified in the annotation (issue type) or what issue caused the participant's confusion with issue types including: incompleteness, fragmentation, incorrectness, poor code example, and ambiguity. Two of the authors independently coded the annotations and reached 75% agreement when coding the issue annotations and 73% for the question annotations – remaining annotations were discussed until agreement was achieved.

In the annotation reading condition, we analyzed how often participants said that an annotation was helpful or unhelpful in order to better understand what annotations succeeded in helping participants. We calculated average helpfulness by how many participants said an annotation was helpful and dividing by how many participants encountered the annotation. We ranked annotations from most helpful to least helpful by how many participants said the annotation was helpful subtracted by how many said it was unhelpful.

In the control condition, we kept track of whether and how the participant chose to relay their information to their teammates. We also referenced the auto-generated transcripts to find and count whenever a participant stated a question.

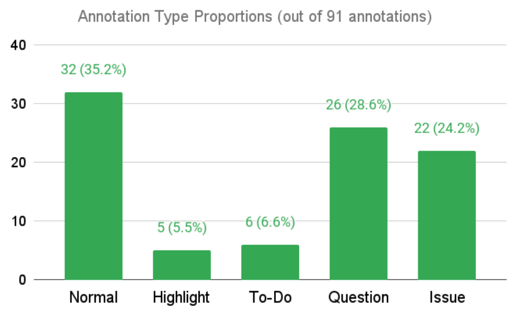


FIGURE 3.5: The proportions for each type of the annotations made in the authoring condition (out of 91), with the exact count for each type above the bar and the proportion in parentheses.

3.4.2 Results

On average, participants in the control completed 1.5 of the 4 steps, authoring participants completed 1.4 steps, and the reading condition completed 2.5 steps (see Figure 3.4). Participants in the reading condition performed significantly better than participants in the control and authoring conditions (paired T-test versus control, $p < .01$, paired T-test versus authoring, $p < .01$). This provides evidence annotated documentation helps developers in using API documentation.

In the control condition, 1 participant chose to take notes in a Google Doc and 1 participant made comments in their code as notes for their future teammates. 2 participants spoke aloud when they had a thought that they would want to share as a note to their teammates, but did not actually write any notes down. The remaining 6 control participants did not take notes or verbally indicate the intent to take notes at any point during the study. This suggests that without a mechanism for externalizing their thoughts, these 6 participants may not have been able to actually share what they learned and only 2 participants had any artifact to share with their future teammates.

How Developers Annotate Documentation

The 10 participants in the annotation authoring condition created 91 annotations across all five of the annotation types (see Figures 3.5 and 3.6). The annotations that participants authored were, on average, relatively short in length at 9.31 words (minimum = 0, maximum = 34, median = 8). On average, each participant authored 9.1 annotations (median = 8, standard deviation = 4.094, minimum = 5, maximum = 18), with the most used annotation type being the normal-type at 32 authored annotations (35.1%). 2 participants in the annotation *reading* condition created 6 annotations (3 normal, 2 highlights, and 1 question), resulting in 97 annotations across all conditions. The rest of the analyses just look at the 91 annotations from the authoring condition.

Considering the large amount of normal type annotations and how normal annotations can contain nearly any type of information, we sought to characterize the content of these annotations. Through open coding, two coders defined 5 categories – "note to self" in which the participant made a note about the documentation's content that was primarily for themselves, "explanation of code" in which the participant tried to better explain what a particular code example was doing, "hypothesis" in which the participant guesses about how some part of Piling works, "important to task" in which the participant highlights a particular part of the documentation as critical for one of the steps of the task, and "other" for any annotations that did not fit into the previous categories. With this categorization, we had 12 "note to self" annotations, 10 "explanations of code", 7 "hypotheses", 2 "important to task" annotations, and 1 "other" annotation. The 1 "other" annotation was an annotation with no content that was created purely as a navigational aid. Participants commonly used "note to self" to keep track of their information and contextualize it to their open task, and used "hypotheses" as ephemeral thoughts about the documentation – these activities are more directly supported in our tool, Catseye, within the context of the code.

Unexplained or poorly explained code examples are a frequent problem in documentation [4, 163] and Piling was no exception, so our participants attempted to explain some of the code examples and, sometimes, contextualize them to the goals of the task. The most helpful and second most helpful annotations are both explanations of code with the most helpful explaining how to use Piling's row property to create columns, and the second most helpful annotation explaining how the code example for `piling.arrangeBy` works and how to adapt the code example to work using a callback function – both necessary steps for completing the task. Further, in our qualitative coding of issue and question annotations, poorly explained code examples were the most common type of identified issue, suggesting that participants valued the ability to express lightweight thoughts about code in context.

Developers also had many questions that relate to documentation issues reported by prior studies [4, 163]. Ambiguity and poor code examples were the source of the majority of developers' questions, which matches the findings reported in [163], with ambiguity, in particular, standing out as a common and severe blocker for developers. Ambiguity and fragmentation issues also resulted in questions that were difficult for annotation authors to answer, with only 33% of questions caused by ambiguity and 33% of questions caused by fragmentation being answered. Considering some participants were able to solve fragmentation issues using multiple anchors with Adamite, this suggests these developers' questions may have been answered if they had been presented with these annotations. In fact, in the Adamite reading condition, 2 participants had their issue of `aggregateColorMap` not compiling solved by an annotation that used multiple anchors to link to the part of the documentation that defines `aggregateColorMap`.

Considering roughly half of the authored normal annotations are primarily beneficial to the original author (i.e., notes to self and hypotheses) and every participant made a personal annotation (i.e., notes to self, hypotheses, to-do's, and highlights), we find evidence that annotating is an effective mechanism for externalizing information and helpful for the author. Further, all participants revisited at least one of their annotations at least once (min = 1, max = 36, average = 10.225 revisits per participant), suggesting participants were able to get some utility out of their annotations. We also included 6 notes to self and 3 hypotheses in the reading condition to see whether these thoughts could benefit other developers. Notes to self, in particular, were the most revisited type of annotation by their authors, and participants, on average, revisited these notes 1.8 more times – more than any other annotation type or coded normal annotation types.

Some of these annotations were used in conjunction with Adamite's other novel features, resulting in the annotations being more useful. The most commonly revisited annotation, a note to self, had 5 anchors with each anchor describing a necessary step in order to properly instantiate the `piling` object. The participant pinned this annotation such that they could reference the anchor steps in their CodeSandbox project (which was open in a separate Chrome tab) – this annotation was also useful

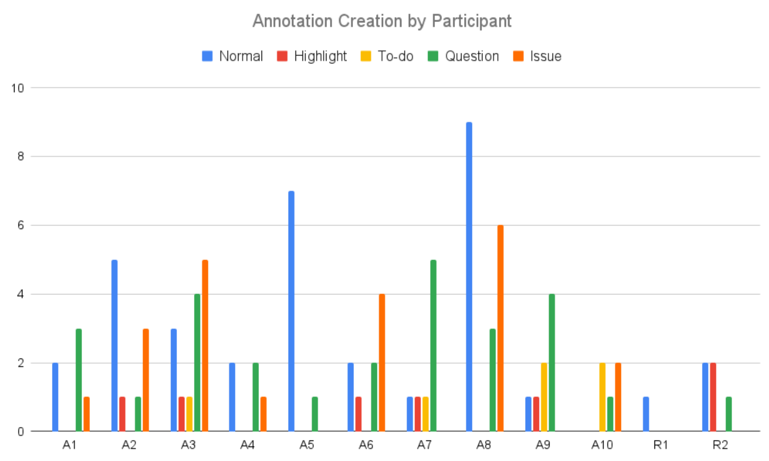


FIGURE 3.6: Which participants made what type of annotation. A1 through A10 refer to the 10 authoring condition participants. R1 and R2 are the two reading condition participants who created annotations.

in the annotation reading condition with one participant replying to thank the author. This shows that Adamite's features not only support the creation of lightweight notes but also allow developers to utilize their and other people's notes in context.

In terms of task completion, participants in the annotation authoring condition on average completed 1.4 steps out of the 4 steps required to complete the task (standard deviation = 0.69, minimum = 1, maximum = 3, median = 1) (see Figure 3.4). There is no statistically significant difference between the control condition completion rate of 1.5 and the 1.4 in the authoring condition (two-tailed T-test, $p = 0.78$), suggesting that annotating the documentation, while not increasing their performance, also did not require so much overhead that participants were unable to complete the task in the same amount of time as if they had not been annotating. Moreover, considering how often developers' revisited their notes, specifically their "notes to self", this suggests authors were able to successfully use annotations as an externalization of their thoughts.

How Developers Use Annotated Documentation

Participants in the reading condition read, on average, 23.7 annotations (including revisiting annotations they had already read, with 72% of annotations read more than once), and read, on average, 15.6 unique annotations. Participants found 45% of the annotations that they encountered helpful, and only 8% not helpful. The top-performing 6 participants in the reading condition also reported the highest proportion of helpful annotations, suggesting that their success may be attributed to the successful use of the annotations.

Of the 32 annotations included in the reading condition, the most helpful type of annotation for readers of the documentation was answered question annotations, with, on average, 54% of participants who encountered them stating they were helpful. Normal-type annotations were the second-most helpful type of annotation (avg. 47% helpful) and issue type annotations helped on average 35% of the time. Some issue type annotations were more helpful than other issue annotations including annotations that identified poor code examples, which were helpful, on average, 45% of the time. Even though these issue annotations did not necessarily suggest a solution, they did work in confirming the participant's suspicions that the documentation itself was incorrect and not the participant's implementation. Sometimes, participants found useful annotations through search – participants searched a total of 78 times and 11 of these searches returned an annotation that the user immediately found useful.

Participants especially appreciated the normal type annotations that were explanations of code, with participants finding them helpful 63% of the time. Code explanations typically elucidated what a code example was illustrating or explained how to adapt a particular code example for the purposes of the task. "Notes to self" were also surprisingly useful, with participants finding them helpful 53% of the time – given that the notes to self typically represented a thought or reminder the developer had about the documentation while completing the task, these results suggest that the participants in the reading study had similar thoughts about the documentation. Conversely, hypotheses were not very helpful, with only 16% of participants finding them helpful – given the uncertainty of these annotations, participants may have found them less trustworthy. These results suggest that explanations of code and developers' personal notes can be useful if they are framed in a knowledgeable fashion, while hypotheses are more useful for the original author.

3.5 Discussion

Our results suggest that annotated documentation is useful for documentation readers in overcoming some of the known barriers of documentation and that the act of annotating when learning a new API can help developers keep track of their thoughts and open questions. Creating annotations was also useful to the author as a form of self-explanation, which has been shown to be useful for learning in prior studies [23, 30], and these self-explanations, or "notes to self", were useful to others. The novel features of Adamite, especially types, multiple anchors, and pinning, helped annotation authors better structure and contextualize their information and helped annotation readers find relevant information.

In the creation and evaluation of Adamite, we sought to explore to what extent annotations may help annotation authors and readers in overcoming previously-reported shortcomings of documentation. Through this exploration, we have evidence developers are able to identify documentation issues using annotations and are able to answer some of their documentation questions. Specifically, our participants were able to identify "poor code examples", "incompleteness" and "ambiguity" issues, two of the largest blockers when using documentation [163]. Other developers can make use of these answered questions and issue annotations, with answered questions as the most helpful annotation type and explained code examples also helping annotation readers. However, annotations cannot solve every documentation issue. If the API and its documentation are updated, the annotations may go out of date, at which point they may be more harmful than helpful. While our algorithm attempts to reattach the annotation to its anchor point, the annotation content will not change to reflect that reattachment, at which point the content may be incorrect. This tension when updating content between developer-authored meta-information and the original source it is connected to is further explored with our tools Catseye and, in particular, Sodalite.

Chapter 4

Catseye: Meta-Information for Sensemaking About Code

This chapter is adapted from my paper:

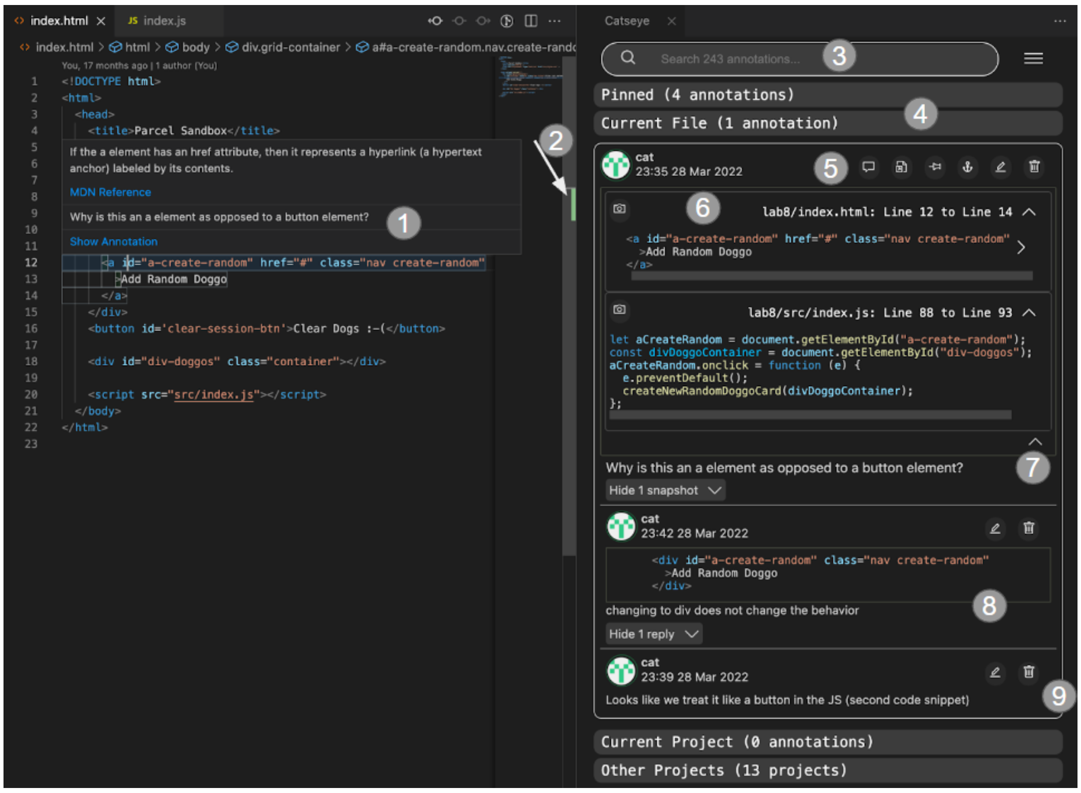
[68] Amber Horvath, Brad A. Myers, Andrew Macvean, and Imtiaz Rahman. 2022. "Using Annotations for Sensemaking About Code". In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA*. ACM, New York, NY, USA, 16 pages.

4.1 Overview

In our previous tool, Adamite, we found evidence that developer meta-information, in the form of annotations, was useful in overcoming known barriers in documentation usage. In particular, *later* users of the documentation experienced a significant increase in task completion. However, the *initial* authors did not experience a significant effect. Despite this, we had evidence that developers did appreciate the ability to annotate programming-related documents with their thoughts, questions, and hypotheses that they wanted to keep track of. With this in mind, we wanted to explore how to make annotating more beneficial for the initial user.

One context in which keeping track of information is possibly even more important than in the context of using developer documentation is when actively programming. Prior work has discussed some of the challenges developers must face when making sense of code. A developer must maintain their own task context [112], while also keeping track of the various questions [146] and hypotheses [106] they have about the code, the answers they find to these questions [91, 146], their code locations of interest (commonly referred to as the "working set" [18, 32, 82, 174]), the different versions of the code they try [75, 172], and how those various versions produce differing outputs [75]. Strategies for keeping track of these various forms of information are rarely successful and are often siloed into different tools or approaches, given the different types of information to track.

In this chapter, we explore the concept of generalizing annotating for capturing different forms of meta-information using a singular, unified mechanism in the IDE with our tool Catseye. Catseye adopts and extends many of the features introduced in Adamite, such as multiple anchors and pinning, while adding in support for programming-specific information tracking, including lightweight versioning and output capturing. Catseye has advantages over other forms of information tracking commonly employed by developers, such as code comments for tracking open


index.html
index.js
index.html > html > body > div.grid-container > a#a-create-random.nav.create-rand
You, 17 months ago | 1 author (You)
Catseye
Search 243 annotations...
1
2
3
4
5
6
7
8
9

```
<!DOCTYPE html>
<html>
  <head>
    <title>Parcel Sandbox</title>
```

If the a element has an href attribute, then it represents a hyperlink (a hypertext anchor) labeled by its contents.

MDN Reference

Why is this an a element as opposed to a button element?

Show Annotation

```
      <a id="a-create-random" href="#" class="nav create-random"
        >Add Random Doggo
      </a>
    </div>
    <button id='clear-session-btn'>Clear Dogs :-(</button>

    <div id="div-doggos" class="container"></div>

    <script src="src/index.js"></script>
  </body>
</html>
```

Pinned (4 annotations)

Current File (1 annotation)

cat
23:35 28 Mar 2022

lab8/index.html: Line 12 to Line 14

```
<a id="a-create-random" href="#" class="nav create-random"
  >Add Random Doggo
</a>
```

lab8/src/index.js: Line 88 to Line 93

```
let aCreateRandom = document.getElementById("a-create-random");
const divDoggoContainer = document.getElementById("div-doggos");
aCreateRandom.onclick = function (e) {
  e.preventDefault();
  createNewRandomDoggoCard(divDoggoContainer);
};
```

Why is this an a element as opposed to a button element?

Hide 1 snapshot

cat
23:42 28 Mar 2022

```
<div id="a-create-random" class="nav create-random"
  >Add Random Doggo
</div>
```

changing to div does not change the behavior

Hide 1 reply

cat
23:39 28 Mar 2022

Looks like we treat it like a button in the JS (second code snippet)

Current Project (0 annotations)

Other Projects (13 projects)

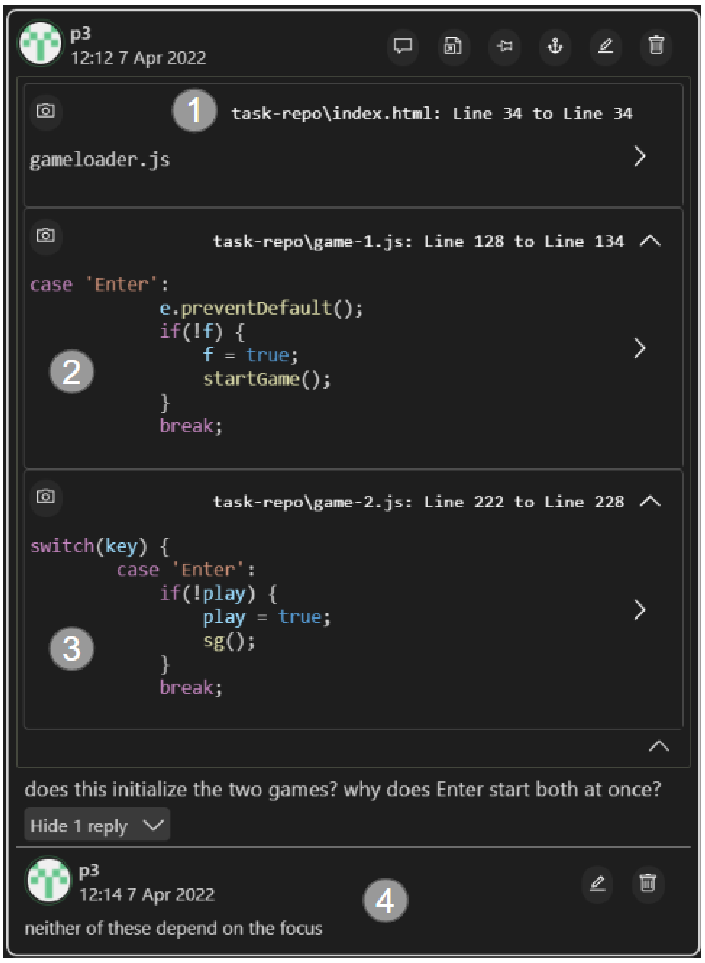
FIGURE 4.1: Catseye as it appears in Visual Studio Code. (1) shows how the annotation appears in the editor – the code is highlighted with a light gray box and, when hovered over, the annotation content appears in the pop-up with any other documentation. Clicking on the "Show Annotation" button opens and brings into focus the Catseye pane if it is not already visible, then scrolls to the annotation. (2) shows the annotation location(s) in the scroll bar gutter in a light green. (3) is a search bar for searching across the user's annotations. (4) is the Catseye pane – the pane is segmented into sections corresponding to the annotations' locations in the file system, with the "Current File" section currently open. (5) is an annotation – the top of the annotation shows the author and creation time information on the left, and buttons for various actions. (6) shows the two code anchors for the annotation. (7) is the content the user added as an annotation to the code snippets. (8) is a snapshot of the code at a previous version with a comment added by the author about this version of the code. (9) is a reply to the original annotation.

tasks [153], by both allowing for a higher level of specificity through anchoring the meta-information to the relevant code yet abstracting that information away from the source code, such that it does not clutter the source with potentially erroneous information. Our lab study suggests that Catseye did assist in information tracking, with participants using Catseye performing better on a debugging task when compared to participants using their own information tracking strategies.

4.2 Catseye

4.2.1 Overview of Catseye

We developed Catseye– an extension for the Visual Studio Code editor [110] – that allows developers to keep track of their tasks, open questions and hypotheses, answers to these questions, and more in the form of annotations attached to one or more snippets of code (see Figure 4.1). We chose to adopt some of the features of our earlier Adamite system [65] for Catseye, given Adamite's focus on supporting developers' information tracking on the web through annotations. Adamite showed



p3
12:12 7 Apr 2022

task-repo\index.html: Line 34 to Line 34

```
gameloader.js
```

task-repo\game-1.js: Line 128 to Line 134

```
case 'Enter':
            e.preventDefault();
            if(!f) {
                f = true;
                startGame();
            }
            break;
```

task-repo\game-2.js: Line 222 to Line 228

```
switch(key) {
        case 'Enter':
            if(!play) {
                play = true;
                sg();
            }
            break;
```

does this initialize the two games? why does Enter start both at once?

p3
12:14 7 Apr 2022

neither of these depend on the focus

FIGURE 4.2: An annotation made by a participant in our study. (1), (2), and (3) show the 3 different code anchors the participant created across multiple files, with the first anchor ("gameloader.js") as the site of their question, and the remaining two anchors and reply (4) answering their question. The annotation was pinned.

the benefits of multiple anchors and pinning, and we expected that these features would help with code comprehension issues, such as managing a working set. We also introduced novel annotation features for Catseye, such as code snapshots for micro-versioning, to help with other information tracking needs.

To create an annotation, a developer selects a snippet of code in the editor and, using a keyboard shortcut, the context menu, or Visual Studio Code's Command Palette, indicates that they want to create an annotation. The Catseye pane will update with a preview of the annotation, where the developer can add text and choose whether or not to pin the annotation. Once the annotation has been created, it will appear in the Catseye pane and the editor will update with a light gray box around the annotated code at the anchor point (see Figure 4.1 at 1). With an annotation, a developer can click on it to jump to the anchor point in the code (and vice versa), build upon it through adding additional code snippets as "anchors", capture versions of the code and the code output, "pin" the annotation for easier navigation, "reply" to the annotation with more information, search for the annotation, export the annotation as a code comment, and edit and/or delete the annotation.

Given our high-level goal of creating an annotation system where annotations serve as ephemeral notes when making sense of code, we explicitly designed annotations to function similarly to Google Doc comments or Microsoft Word comments.

Annotations in Catseye move around with the code as the code and its location in the editor change over time, and the annotations appear in their own designated area that is detached from the developer's editor. We chose this design metaphor to emphasize the point that annotations are not code comments – they are separate, meta-level notes that are attached to but abstracted from the developer's working context. Annotation anchors update whenever the developer edits their code, and the copy of the code at the anchor that is shown in the annotation (Figure 4.1 at 6) is updated whenever the developer saves their code.

Annotation code anchors can also be used as navigational aids. The developer can click on the code or the file path in the annotation (Figure 4.1 at 6 and Figure 4.2 at 1) which will open that file in a new tab if it is not already open, bring that file's tab to the front if it was not already, and scroll to the code's location in the file. Additional navigational affordances are provided for pinned annotations – a developer can use a keyboard shortcut to cycle through each pinned annotation's location to help with navigating through important code patches. Figure 4.2 shows a pinned annotation one participant in the user study created to help with managing their working set.

Given the mutable nature of code, keeping annotations attached introduced some design challenges unique to Catseye in comparison to other annotation systems designed for more static information. Since code is expected to change, retaining the original anchor point becomes more important as the annotation's content is more likely to become out-of-date – we choose to store a copy of the user's original anchor point for reference as the code changes. The developer can also explicitly save a version of the code by clicking the snapshot button on the code anchor box (see Figure 4.1 at 6). Once the snapshot has been created, the developer can edit the snapshot to add metadata, such as what output that version of the code produced (Figure 4.1 at 8).

Another design challenge is how to handle the case where the user deletes the code that an annotation is attached to – should the annotation be removed as well or should it persist? Different annotation systems do different things – Overleaf comments will persist when the anchor is deleted, with a line showing where the text used to be, while Google Doc comments will be removed if their anchor is removed. We chose to follow Google Doc's design choice, with the rationale that, if the user wants the annotation content to persist, they can attach an additional anchor to the annotation or export the annotation as a code comment.

A related issue is what to do with annotations on code which is copy-and-pasted. Again, other annotation systems do different things. We decided to *not* copy the annotation with the code with the justification that we want to reduce as much potential annotation clutter as possible (thus we choose to never create an annotation without the user's explicit request). These design choices also turned out to be the most useful way for these features to work in the my personal usage of the tool (see Section 4.6). Notably, handling copy-pasted code is handled more directly with our later tool, Meta-Manager, where the relationship between the copied code and pasted code is retained and tracked. In the proposed work, which involves integrating Catseye and Meta-Manager, unifying these two different design goals into a comprehensible interaction will need to be addressed.

Oftentimes, when a developer wants to keep track of some information, they will want to resolve or build upon the information they initially felt was worth jotting down. Catseye follows a similar model to other annotation systems where "following up" on the content of a note is kept very general, so the developer can either edit their original note, or reply to it with their additional thoughts.

4.2.2 Background and Design Goals

In creating Catseye for Visual Studio Code, we were particularly interested in helping developers capture and keep-track of their ephemeral thoughts, questions, concerns, and open action items related to their code, since this is the least well addressed aspect of previous tools. We envision that annotating will help with the following use cases:

- **Keeping track of developers' questions and hypotheses about code.** Sillito et al. [146] found that, during software maintenance tasks, developers reported over 40 different kinds of questions they had about the code and that there is little tooling support for finding answers to those questions. They also found that there is limited tooling support for helping developers keep track of these questions as they come to an answer. Given that Catseye 's annotations retain the original context of the code, and allow for composition of multiple code snippets through multiple-anchoring, we hypothesize that annotations may help with keeping track of these complex questions and the eventual answer a developer finds.

- **Keeping track of facts developers learn about code.** During any task that requires comprehending code, developers will naturally collect a body of knowledge about the code base [91]. Only some of these facts are appropriate as documentation, either because the behavior of the code is expected to change (such as during debugging) or because there is uncertainty about the veracity of the fact. We hypothesize that Catseye and its annotations will help with externalizing these thoughts while not requiring laborious clean up, since the source code is unaffected.

- **Keeping track of developers' open to-do items.** In trying to complete a complex programming task, a developer needs to keep track of a multitude of both high level goals and lower-level implementation steps in order to achieve that goal which the developer may forget, especially when interrupted [123]. Developers can use annotations to mark the code to change with the details of their "todo" item, compose snippets that are related to the change using multiple anchors, and can pin and un-pin the annotation as a way of marking whether or not the task still needs to be addressed.

- **Helping developers navigate their code.** An oft reported difficulty in programming is navigating the code base, especially when it is large. Developers typically discover a "working set" of task-relevant code fragments [18, 32, 82], then spend time navigating among these fragments as they implement their change. This navigation takes up a large amount of time, especially since these fragments can be difficult to return to [82]. We expect that clustering annotation anchors using multiple anchors, pinning these annotations for easier tracking, and using the code anchors as quick links will make this navigation easier.

- **Keeping track of localized changes.** When a developer is implementing a change, they often try multiple versions of the code in order to investigate the differences in output and ensure that the change works. These changes can be relatively small (i.e., less than 5 lines of code), may not be tracked in version control [75], and switching between these versions can be difficult if the prior versions are not retained, especially since they may be inaccessible through undo commands [172]. Catseye allows developers to snapshot their code for versioning, such that developers can keep track of the different changes they try and can optionally associate these versions with the output they produced.

- **Keeping track of changing system output.** While testing changes, developers have reported a need for keeping track of what version of their code produced what output [75] and have used strategies such as copy-pasting the output into text files. We provide annotations as a place to store these outputs – developers can either reply to their annotation with the output values or edit their snapshots with the output which comes from that version of the code, thereby leveraging the context of the code.

Notably, the use cases described above are designed for the benefit of the *original author of the annotation*. We chose to focus on supporting the initial annotation author given the goal of supporting a developer's tracking of information, which is largely localized to a single author and their implementation session(s).

4.2.3 Implementation Notes

One particularly important aspect of Catseye is managing the code anchors as the user actively edits the code. Annotation anchors are kept up-to-date using Visual Studio Code's document change event handler. Whenever the user modifies a file that contains an annotation, the Visual Studio Code API generates a change event object that we interpret. For simple cases, such as adding a new line at the top of the file, updating the anchors is trivial, but, in the case of more complex changes, such as pulling in a new version of code from GitHub (which the Visual Studio Code API treats as many small edits applied in rapid succession), using the event API can fail, resulting in an incorrect anchor point. If this occurs, we delete the annotation. Given that tracking source locations has been a long-standing challenge in software engineering [134], we leave more robust methods for fixing up broken anchors to future work, along with investigating the design space of how annotations should be updated or archived given these large changes. In fact, Sodalite expands upon the code anchoring functionality thru introducing new types of code anchors that are designed to be more flexible, such that un-anchoring is less common (see Chapter 5).

4.3 Lab Study

In order to understand how developers keep track of information while making sense of code when using their own strategies and when using annotations, we ran a small lab study. Participants in the *experimental* condition authored annotations while using Catseye to help them keep track of information, while participants in the *control* condition used whatever strategies they normally would employ. The lab study consisted of a training task, then a debugging task, and ended with a survey to assess the participants backgrounds, their experience with Catseye if in the experimental condition, and their experience completing the task. We chose to use a between-subjects design as opposed to a within-subjects design due to the nature of the task. The study took around 90 minutes, so adding another training session and 45 minute task would make the study too long. Further, as discussed in [84], since these are problem-solving tasks that you can only do once, creating 2 tasks which are independent but of equal difficulty is challenging.

Game	Bug	Minimal Solution
Both	Unable to Play Games Independently	Change one of the event listeners to a different key (1 value change)
Snake	Screen Does Not Refresh	Adapt Tetris's screen clearing function to Snake (10 line change)
Snake	Snake is Too Fast	Adapt Tetris's timing function to Snake (10 line change)
Snake	Snake is Drawn Incorrectly	Change the constant value for the snake segment length (1 value change)
Snake	Food Collision Check is Incorrect	Change the ORs in the boolean to ANDs (2 value change)
Tetris	Blocks Falls in Last Key Press Direction	Set current direction of block fall to "down" on each game loop (3 value change)
Tetris	Rotating Square Causes Square to Move Upwards	Add conditional to prevent square from being rotated (3 line change)
Tetris	Game Does Not End	Change conditional to whether the stack of blocks is at the top of the screen (1 value change)
Tetris	Game Calculates Score Incorrectly	Increment user's number of cleared rows instead of setting to last clear row value (1 value change)

TABLE 4.1: The bugs present in the two games. "Value" refers to a construct in the program, such as an operator, boolean, or variable.

4.3.1 Method

Training Task

Both conditions included a training task using a repository of website templates[1] to either familiarize the participants with Catseye (experimental condition) or to showcase how the participants currently keep track of information when programming (control condition). Participants in the Catseye condition learned how to create and edit an annotation, pin and reply to an annotation, navigate and version their code using annotations, and collect system output, with all functionalities contextualized to how they may be useful for keeping track of different kinds of information. Participants in the control condition were asked to describe how they currently keep track of the different types of information we expect Catseye to support. In this way, we tried to make sure that both groups were primed about the kinds of activities that Catseye is designed to support.

Main Task

For the main task, participants were instructed to understand and attempt to debug a website. Participants were told to imagine that they were a new developer on a team and that they were tasked with understanding and debugging some code. For the first 15 minutes, the participants were not allowed to edit the pre-existing code (but they could add comments and print statements) as part of the scenario in which they are new to a team and should spend time familiarizing themselves with the code prior to contributing changes. This also allowed us to investigate differences in the kinds of annotations made while understanding versus debugging and editing. After the 15 minutes of understanding and testing the code, participants had 30 minutes to attempt to use what they learned to resolve issues they had discovered.

The website included buggy implementations of Snake and Tetris. Each game had 4 bugs, with an additional bug that affected both games, totalling 9 bugs (see Table 4.1). We chose these two games since they are both relatively well-known, are event-based which makes understanding their structure less straightforward, and have clear requirements such that testing the games takes less time in comparison to actually debugging their logic. Similar tasks have been used in related studies [123, 124].

[1]https://github.com/ShauryaBhandari/Website-Templates

What Information to Track	What Aspect of the Task	Explanation
Questions, Hypotheses, and Answers	Debugging task	Debugging naturally leads to many questions and hypotheses about the program behavior but subsequent answers may be lost or forgotten [146]
Facts	Poorly-written and documented code	Developers are tasked with learning what the code constructs are and how they are used
Open Tasks	15-30 time split	Allow developers to discover many bugs, then have them decide which ones to focus on and how to fix them
Navigation	Poorly-organized code	Constructs, including methods and classes, are spread across multiple files including some files the participant cannot edit
Localized Changes and Output	Arcade Games	By having two arcade games, participants are tasked with making changes and seeing how that affects each game and how that affects each game's output

TABLE 4.2: How the study task encapsulates the types of information Catseye supports.

The code was specifically designed to be confusing in order to make keeping track of information particularly important (see Table 4.2). Given prior literature around what makes code confusing [146], we purposefully included bad code smells such as poorly-named variables, global variables, lack of organization amongst methods, and no documentation. Since participants only had 45 minutes for the task, we wanted to necessitate keeping track of information while also keeping the task semi-realistic through using known issues when comprehending unfamiliar code. To further validate the realism of the code, we included two questions in our post-task survey that asked participants how similar the code they saw in the study is to code they have encountered during their time as developers and how frequently they have encountered such code. Participants reported the code is similar to code they have encountered before[2] but that they do not encounter code like this very frequently[3].

4.3.2 Participants

We recruited 13 participants (5 women and 8 men) using study recruitment channels at our institution, and advertisements on social media. Participants were randomly assigned between the control and experimental conditions, with 7 participants in the experimental condition and 6 in the control condition – participants in the experimental condition are referred to as "P1" through "P7" and control participants "C1" through "C6".

All of the participants were required to have some amount of experience using JavaScript, to have a GitHub account, and to regularly use Visual Studio Code. The participants' professions included graduate students in computer science-related fields, undergraduate students in computer science, and professional programmers. On average, participants had 10.2 years of programming experience, 5.2 years of professional programming experience, and rated their familiarity with JavaScript at 4.5 out of 7. Participants in the control condition had more experience and more professional experience, on average, than experimental participants, but not significantly more.

4.3.3 Analysis

Across both conditions, we objectively coded what bugs the participant succeeded in fixing (see Table 4.1). In the experimental condition, we analyzed the video recordings and log data to count how many annotations each participant authored and

[2]average = 3.4 out of 5, using a 1-to-5-point Likert scale from very dissimilar to very similar
[3]average = 2.6 out of 5, using a 1-to-5-point Likert scale from never to very frequently

how often they interacted with their annotations to assess the utility of the annotations for keeping track of information. We additionally logged whether or not any annotations were made in the first 15 minutes and, for annotations created during the debugging part of the task, what bug the participant was attempting to solve at the time of creation. We analyzed the videos in the control condition to log the same types of interactions including the artifacts developers made in that condition, such as code comments and external notes.

We additionally labeled the annotations and control condition notes with the type of information it was being used to help keep track of. We objectively coded this conservatively based off the content. If an annotation's or artifact's content was phrased as a question or had a question mark, it was coded as a question; if the content had words such as "might" or "seems like", it was coded as a hypothesis; if the content was phrased as an objective such as "change this", it was coded as a task; and if the content was stated as a fact (e.g., "game-1.js is snake") it was coded as a fact (even if the fact was incorrect). The same process was used for annotation replies.

We counted items as used for "versioning" when they either contained a snapshot (annotation) or were used to mark a change they made to the code base (annotation or control artifact). For navigation, we counted an annotation that is pinned and/or had multiple anchors as used for navigation.[4] We counted an annotation or control artifact as being used for output if the participant used it to store or comment upon the game output. If the content of an annotation or artifact did not fit into any of these categories, it was marked as "Other". For replies, we also labeled whether or not a reply served as an answer to their question annotation – a reply was considered an "answer" if its content was a direct response to the question's content that supported or refuted it.

4.4 Results

Participants in the experimental condition fixed, on average, 1.85 bugs (min = 0, max = 4), while participants in the control condition fixed 0.67 bugs (min = 0, max = 1), a significant difference (*two-tailed T-test, p = .04*). To further explore these results, we investigate what types of information participants chose to keep track of through annotations versus what information control participants used their artifacts to keep track of, how participants used their information when completing the debugging tasks, and how participants performed on the debugging task.

4.4.1 What Information Do Developers Keep Track of with Annotations and Artifacts?

Experimental condition participants created 84 annotations, with each of these participants creating, on average, 12 annotations (min = 6, max = 21, median = 10, std. dev. = 5.446). 44 of the annotations were made in the first 15 minutes and 40 were made in the last 30 minutes. The size of the annotations averaged 12.2 words (min = 1, max = 45, median = 12.5) and they were attached to code averaging 29.3 characters. Each anchor was, on average, 1.59 lines long, with the majority of annotations attached to one line or less of code (71/84).

[4]Since multiple anchors and pinning are unrelated to the text content of an annotation, this means an annotation could be marked as both "navigation" and, for example, "fact".

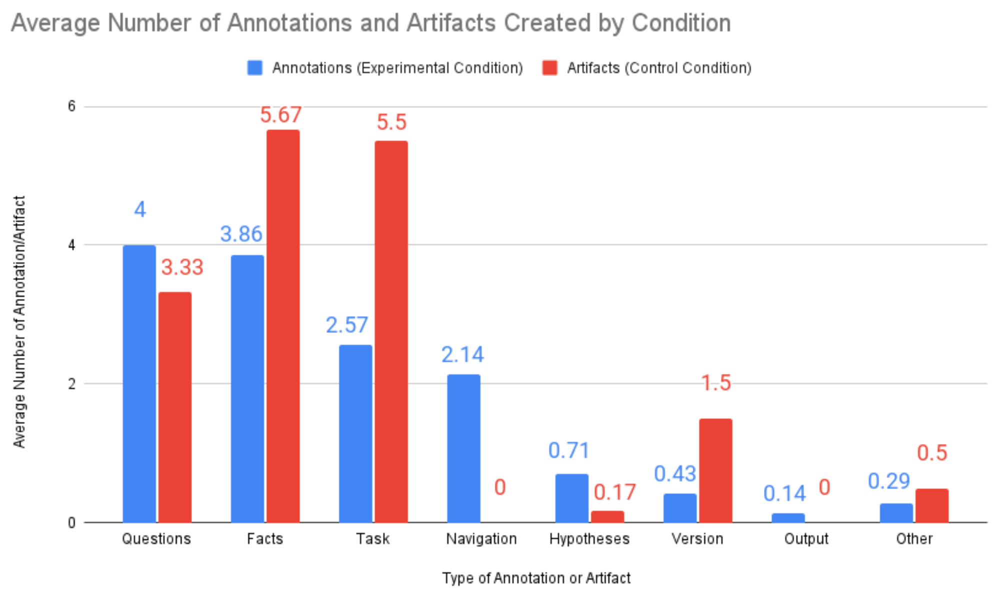


FIGURE 4.3: The average number of annotations and artifacts participants created during the study.

Across those 84 annotations, developers had a variety of types of information they chose to keep track of through annotations (see Figure 4.3). The most common usage for an annotation was to keep track of open questions developers had with 28 out of the 84 annotations being questions (33.3%). 16 of these questions were made during the first 15 minutes, while the remaining 12 were created in the last 30 minutes. 6 out of those 28 questions were definitively answered, while 2 had follow-up hypotheses given program output behavior, and 1 had a follow-up question associated with the original question, resulting in 9 out of the 28 questions being followed-up on in some way. Given the complexity of the task, the amount of answered questions is not particularly surprising, but the fact that participants followed-up on their questions at all provides support for the argument that annotations can serve as dedicated spaces for these questions. In contrast, as discussed below, only 1 of the control condition's 17 questions were followed-up on or answered.

The second most common type of information experimental condition participants kept track of in their annotations were facts they discovered about the code, with facts comprising 27 out of the 84 annotations (32.3%). 23 of these 27 facts clarified information that was explicitly designed to be confusing. For example, P1 annotated `const c = document.getElementById('t')` with "this is the canvas of the tetris game". 18 of these 27 annotations were made in the first 15 minutes, when participants were reading through and understanding the code.

Experimental condition participants also utilized annotations to keep track of their open tasks (21.4% of their annotations) and to navigate the code (17.9% of their annotations). These annotations typically served as reminders to the participant about places in the code base they suspected were related to the bugs they identified in the code. The majority of task annotations were made during the 30 minute debugging phase (12/18) suggesting that there was more of a need for keeping track of their areas of interest in the code once they were developing, as opposed to when they were trying to understand the logic.

Experimental condition participants did not use their annotations for keeping track of their code versions, with no participants using the snapshot feature. Participants did create annotations to comment on parts of the code they added or modified, with 6 annotations made on the participant's own code that they added. Considering that participants, in general, did not edit the code very much, since the bugs did not require large modifications to fix, there may have been less of a need to keep track of small localized changes. Further, the code that participants chose to annotate was usually code that the participants did not edit, with only 12 of the annotations' corresponding code being edited. 3 annotations were used to keep track of output – the small number of output annotations may also be due to the minimal amounts of changes participants made to the code. Further, since the program was a computer game, much of the output changes were graphical which is not output that Catseye can capture currently.

In the control condition, the participants created a total of 100 different artifacts, averaging 16.67 artifacts per participant (min = 2, max = 27, median = 15, std. dev. = 9.771) – which is slightly more artifacts per participant than in the experimental condition, but the difference is not statistically significant ($p = .76$, T-test). This may partially be due to the fact that control participants were primed to think about and show how they keep track of information. Further, all of the control participants had some note taking strategy that they described using in their daily work.

Their artifacts included 78 code comments, 14 external notes (with 5 of the notes being created on a tablet computer, 1 being created in a Notepad document, and the remaining 8 created using pen and paper), and 8 Git commit message[5]. The artifacts averaged 5.95 words (min = 1, max = 22) – notably shorter than the annotations, which averaged 12.2 words per annotation.

The information that control participants chose to keep track of through artifacts differs from the information that was annotated (see Figure 4.3). While, in both conditions, facts, questions, and open tasks were the three most commonly kept track of information types, participants in the control condition kept track of facts the most, while participants in the experimental condition kept track of questions most often. Participants in the control condition also favored "task" artifacts more so than experimental condition participants, with most in the form of "TODO" code comments (66.6%) consistent with prior work [153]. Only one control participant actively attempted to keep track of different versions of their code, and none of the control participants kept track of the output of their code or used any specific mechanisms to help navigate their code, aside from traditional code search (which participants in the experimental condition also used). These results suggest that annotations can promote more entering of questions and answers in comparison to traditional notes and annotations can keep track of other types of information that may otherwise not be captured.

4.4.2 How Do Developers Use Their Annotations and Artifacts?

We quantify usage of annotations or artifacts by counting whenever a user interacted with their annotation or artifact in some way. On average, experimental participants revisited 5.71 unique annotations, and revisited their annotations 11.14 times over the course of the study. In contrast, the control condition, on average, revisited 3.5 artifacts a total of 4.67 times suggesting that the annotations were more successful in encouraging participants to follow-up on their information. One of the two most successful participants, both of whom fixed 4 bugs, also created the most annotations

[5]All of the Git commit messages were created by one participant.

Bug	# of Experimental Participants Who Fixed This Bug	# of Control Participants Who Fixed This Bug	% of Debugging Annotations Made About Bug	% of Debugging Control Artifacts Made About Bug
Unable to Play Games Independently	2	2	12.5%	0%
Snake Screen Does Not Refresh	3	0	12.5%	0%
Snake is Too Fast	2	2	15%	88.6%
Snake is Drawn Incorrectly	1	0	2.5%	0%
Snake Food Collision Check is Incorrect	1	0	7.5%	5.7%
Tetris Blocks Falls in Last Key Press Direction	1	1	35%	5.7%
Tetris Rotating Square Causes Square to Move Upwards	0	0	0%	0%
Tetris Game Does Not End	1	0	12.5%	0%
Tetris Game Calculates Score Incorrectly	0	0	2.5%	0 %

TABLE 4.3: The annotations and artifacts participants created during the study while working on each bug. The experimental condition made 40 annotations while working on bugs, while the control condition created 35 artifacts. The last 2 columns refer to the proportion of annotations made about that bug out of the 40 annotations made while debugging, and the proportion of control condition artifacts made about that bug out of the 35 artifacts made while debugging, respectively.

(21) and revisited his annotations the most, revisiting 14 of them 32 times, suggesting annotation usage may have contributed to his success.

Experimental participants often revisited their annotations to add replies to their annotations, with participants creating 16 replies and 5 out of 7 participants creating at least 1 reply. Replies typically served as an extension of the annotation's original content, with some annotations serving as answers to the original question (6), hypotheses about the behavior of the code (3), and follow-up tasks that they wanted to complete related to the original annotation (3). For example, P5 made an annotation about Snake where the initial annotation just said "Game 1: Snake" and created two replies, with the first reply explaining what the two Snake-related files did, and the second reply listing all of the bugs she had encountered with Snake – she then revisited this annotation 3 times over the course of the study to keep track of her bugs. Replies also sometimes functioned as places to discuss the behavior of the code *after* the participant attempted to fix a bug associated with the annotated code, with 2 replies commenting on whether their implementation worked or not.

Participants also used pinning, multiple anchors, and anchor clicking as a way of supporting their navigation while working on the task. 5 annotations had multiple anchors, 3 annotations were pinned, and the participants used the anchors to navigate the code base 19 times. In contrast, the control condition did not use any of their artifacts to help them navigate the code base.

Three experimental participants also chose to delete their annotations once they were "done" with them, with these participants deleting a total of 14 annotations. The most successful participant deleted 12 of his 21 annotations over the course of the study – whenever he fixed a bug he would find each annotation that related to that bug and delete it, while keeping open the annotations that were still unresolved. His usage of Catseye suggests that annotations can function similarly to comments in systems like Google Docs where, even if the content of the comment is not necessarily a "to-do item", the comments can still be resolved in a similar manner. Control participants only deleted their artifacts, on average, 0.8 times while experimental participants averaged 2.16 deletions, further suggesting that a Google Docs-style design encourages more clean-up than regular code comments or external notes.

4.4.3 How Did Participants Identify and Fix Their Bugs?

All bugs were identified by at least 1 experimental participant and had at least 1 annotation created about it, save for the Tetris square rotation bug. No participants in the control condition identified the Tetris rotating square bug or the Tetris score calculation bug, so no control participants made artifacts about those bugs.

When struggling with difficult bugs, participants seemed to create more annotations and artifacts. For example, the "Snake is Too Fast" bug, which required 10 lines of code to change, was only successfully completed by 3 out of the 7 participants who attempted it, and resulted in the majority of control condition artifacts to be about this bug, along with some annotations (see Table 4.3). Conversely, some of the simpler bugs to fix, such as "Snake is Drawn Incorrectly" had fewer annotations made about them as there was less need for participants to externalize their thought processes.

Some particularly complex bugs led participants in the control condition to utilize their notes in different ways than their experimental counterparts did. 3 control participants made a total of 3 external notes that were either visual diagrams of how they thought the games should function or were algorithmic step-by-step instructions for how to design their bug fix. Since the experimental condition created no similar notes, this suggests that future versions of Catseye may better support users by including a way to attach and create visual diagrams, screenshots, or drawings to annotations and support richer interactions for checking off completed steps in an algorithm.

Participants in the experimental condition occasionally made annotations that documented how they fixed a bug, with 4 annotations created for this purpose. For example, P6 wrote some code to try and fix the Snake Screen Does Not Refresh bug and annotated their code with the text "Attempt at clearing the score" and edited their implementation 3 times to try and achieve the correct behavior. P6 then made 2 more annotations on other code snippets that they were referencing when trying to fix their implementation, hypothesizing about how they could adapt the functionality of that code to solve their bug. Their usage suggests annotations can help with marking and documenting code while debugging, including code the user has added that attempts to fix the bug.

4.5 Discussion

Our experiments lend support to the concept that annotations may be used as a lightweight way of capturing and following-up on information that may not otherwise be kept track of when programming. Participants succeeded in creating questions, following up on those insights, and revisiting these notes in order to fix their bugs and had more success, on average, than the control condition.

Four participants asked to continue using Catseye after the study, with 2 participants creating more annotations in their own code after their session. One of these 2 participants reported back on her usage of the tool in her daily work. She found the tool useful for externalizing her "design-oriented notes-to-self" such as "maybe I should do X instead, if I do decide to do that, this is the code that needs to be edited to make it happen". Notably, this is the type of information she says she would normally write down on a piece of paper and *not* use code comments for since she does not want to create clutter that will only confuse her or her collaborators later. She found Catseye valuable for acting as a space for capturing this "thought history" that leverages the context of the code. Her experience lends further support to our claim that supporting thinking through and keeping track of developers' thoughts in a dedicated space when programming is useful.

Participants found the annotating metaphor familiar and understandable, despite the amount of complex activities participants could use the annotations to support, with participants in the post-task survey saying the system was very easy to learn how to use.[6] Prior work has noted that annotations' flexible nature and structure allows them to be used in a variety of ways [5, 15] – we build upon this by showing that annotations can be used in new ways, including to store output, to store and capture versions of code, and as navigational aids. Typically, attempts to support these different activities are siloed into different research tools; annotation systems show a promising alternative where, by utilizing annotations' flexible nature, they can act as a more general "workspace" for storing and thinking about contextualized information a developer cares about.

Another way that participants used annotations as a general "workspace" for thinking was through using their annotation as a "placeholder" when navigating their code. For example, P3 created a question annotation asking why a certain method gets called twice. They followed-up with a hypothesis stating "seems like it's because it calls the draw function, which has some special logic that only occurs if play is true", but, while writing this reply, they paused to continue exploring the file while reflecting on their hypothesis, before returning to the annotation and finishing their initial thought with a guess, "but you still want to call window.requestAnimationFrame i guess?", given what they had learned while exploring. Three other participants paused while creating their annotations to explore the files and think critically about what they were choosing to annotate, which suggests that the choice to have the annotations in their own dedicated pane separate from the context of the code may better support this kind of self-reflection. Notably, these self-reflections have been shown to improve learning outcomes [31].

Some participants in the Catseye condition thought that the output and version capturing features would be particularly useful, despite not using them in the study. Two experimental condition participants reported in the post-task survey that they would use the snapshot feature in their own programming, since they found it difficult to go back to GitHub to see other versions of their code and they sometimes created many small changes that were not tracked in version control. A third experimental participant noted that they wanted to use this feature to capture output when they are performing maintenance tasks like refactoring and need to keep track of many "moving parts" and how their changes affect the behavior of their code. Since running the study, we enhanced Catseye to automatically capture intermittent output through connecting into the Visual Studio Code debugging API to capture and store run time data as replies when an annotated line of code is run.

Two participants in the control condition and 2 participants in the experimental condition created artifacts that, while phrased as a fact, were incorrect. The control participants added comments above functions incorrectly stating what the functions' purposes were. The 2 experimental participants incorrectly assumed what a function and variable were used for, respectively. These annotations, while incorrect, are only visible to the original annotator, while the code comments could, in theory, be viewed by any collaborator, which could potentially misinform them. Even if the annotations were viewable to collaborators, they would not be in the code acting as documentation, lessening their potential to be harmful. An incorrect annotation

[6]"I consider it easy for me to learn how to use Catseye", average score 6.83 out of 7, with 7 being "Strongly Agree"

could be a learning opportunity with the reply feature, where a collaborator could clarify or correct a misinterpretation of the code.

Two control participants had problems managing their code comments. C1 created a comment noting that a particular part of the Snake code looked like it was used for initialization, then discarded the commit that contained that comment. 10 minutes later, they searched for that comment, forgetting that they had discarded the comment. C2 marked a part of the code with the comment "REVISIT" but then undid a series of changes in order to revert to an older version of the code, removing that comment in the process, and then never revisited that part of the code. Participants in the Catseye condition did not create any code comments (2 participants started to make code comments before removing them and manually converting them into annotations) and did not lose any of their annotations during the study – annotations' meta-nature may serve as a safeguard from erroneously removing them. However, Catseye users could, in theory, lose their annotations by erroneously deleting the code with which the annotation is associated. Currently, Catseye does *not* put annotation creation, editing, or deletion into the Visual Studio Code undo stack, another area where other annotation systems differ – Overleaf similarly does not, whereas Google Doc puts comment creation in the undo stack (but *not* comment editing or resolving!). In my own usage of Catseye, putting annotation creation on the stack would be beneficial, given that I normally annotated code I had just worked on – this means that performing an undo operation is likely to modify the code anchor the annotation corresponds to, which may change the annotation's meaning or cause it to become detached if the code did not exist in the prior edit. Annotation deletion would also make sense to go on the undo stack, especially in the case that the annotation is deleted because its code anchor was deleted, given that an undo operation would bring back that code anchor, thus should bring back the annotation. This is how similar systems, such as Google Docs, works.

4.6 First Author Usage of Catseye

I used Catseye while developing Catseye as a form of "dogfooding." Anecdotally, we report on the annotations created by myself using the same methodology of labelling each annotation by the primary type of information it was meant to keep track of. We omit annotations made without any text content, as they do not have enough context for labeling, and annotations made purely for testing the application, considering they do not represent "real" usage of the tool. All of these annotations were created by myself to help myself, either in the short term (so they were ephemeral) or for when I returned to the code later.

Over 12 months, I created 182 "real" annotations in the Catseye repository across 25 source files, with each annotation averaging 28.95 words (min. = 2, max. = 143, std. dev. = 27.99). 42 annotations had a total of 58 replies, 8 annotations had 18 snapshots, 3 annotations had multiple anchors,[7] and 9 annotations were pinned.[8] 110 of these 182 annotations were deleted as I finished open tasks and iterated over the code, including removing the code the annotation is anchored to which deletes the annotation.

[7] Multiple anchors were added late in development, so the lack of multiple anchor usage is primarily due to the short amount of time to use the feature.

[8] This is a conservative count, considering we used our log data and the log does not count whenever an annotation is pinned or un-pinned, just whether the annotation was pinned the last time it was updated in the database.

The content in each annotation differs slightly from the annotations created in the lab study, perhaps due to the different nature of development. The most common annotation type was "task" type annotations, with 48 of the 182 annotations reporting some open action item I needed to act on – in contrast to the task annotations made in the study, these task annotations served as reminders for places to change when performing maintenance tasks like refactoring, as opposed to parts of the code that may have a bug. I also created many question annotations (47 out of 182) – 12 were replied to, with 4 of these replies answering the original question, and 17 questions were deleted. These questions typically pondered the system behavior, previous design choices, or details of the Visual Studio Code API. The third most common code was "Other", with 16 annotations. All of these "other" annotations served as a reaction to the code, with these reactions sometimes pondering the former design rationale and sometimes expressing frustration with implementation challenges. For example, one annotation, anchored to a particularly confusing function said "This is a pain". No annotations were made like this in the lab study, suggesting that actively implementing and writing code may elicit different types of information than code understanding and debugging tasks.

Actively using Catseye also led to changes in the tool's design. Initially, an annotation would be copied if the developer copied the annotation's code anchor point. However, given that code was often copied to be used as a template, in active development, copy-pasting the annotations resulted in duplicated annotations with irrelevant content as the code changed, which I found to be more distracting than helpful. This experience informed our design decision to not copy the annotation content when the user copy-pastes the anchor code. However, there may be situations in which it would be better to copy the annotation with the code, such as when the annotation serves to document behavior about the code. Future versions of Catseye might benefit from allowing the user to choose whether or not to have the annotation copied with the code, or alternatively, to add a new anchor to the original annotation.

I found the tool useful for externalizing and thinking through problems and stopped using code comments in favor of annotations. The ephemeral nature of the annotations was particularly useful in the development of Catseye since the code was nearly constantly changing which lead to a lot of uncertainty about the design and implementation details – information that I did not want to commit to the code base as comments in case the information ended up becoming obsolete (a common problem with code comments [49, 131, 155, 156]).

4.7 Conclusion and Future Work

In our development of Catseye, we found evidence that note taking and collecting of code-related meta-information was helpful for developers when tracking information. Further, we found evidence that these activities could be supported using a singular, lightweight interaction design – annotation. These findings extended our previous findings from Adamite by generalizing our approach to a different developer sensemaking domain while tracking other types of information. Much of the success of the annotation approach can be attributed to the localized nature of the annotations and their connection to the source material through their anchor point. This connection allows for the note to serve as a navigational way point, allows for commenting on different types of code with the same interaction method (e.g., commenting on a variable within a line of code versus commenting on a whole conditional that spans multiple lines), and supports code-specific activities including versioning and output capture. With that in mind, we sought to extend our code anchoring support and the types of information that may be attached to the code anchor points through extending Catseye into a long-form documentation tool, Sodalite.

Chapter 5

Sodalite: Meta-Information to Support Documentation Management

This chapter is adapted from my paper:

[67] Amber Horvath, Andrew Macvean, and Brad A. Myers. 2023. "Support for Long-Form Documentation Authoring and Maintenance". In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Washington, D.C., October 2-6, 2023. IEEE.

5.1 Overview

The connection between code and text written *about* code has long been studied and explored. From literate programming [35, 81, 145], which has had a recent resurgence with the rise of Jupyter notebooks [77], to various forms of documentation [1, 32], there exists a tension between making code comprehensible through expressing the design intent in a more naturalistic and human-readable form, while allowing for the control that traditional programming languages provide. Our approach with Sodalite[1], sits at the intersection between these two approaches by allowing the source code to be *anchored* to the corresponding written text that seeks to e.g., describe how it functions, or how it came to be.

Once the text is anchored, this relationship can be utilized in new ways. While Catseye and Adamite automatically attempted to update and repair any links between annotation and source text or code that appeared broken, this process sometimes failed and the annotation would be thrown away. This often occurred when the original source document changed in such a way that there was no longer an appropriate anchor location for the annotation, such as during a Git pull or if the API documentation changed between releases. While the fact that the annotation is no longer anchored, thus, may no longer be appropriate for the author or a reader to reference, the fact that it became un-anchored itself may be useful. For example, if I have annotated an API method's parameters with the specifics of my particular implementation, it would be useful to know that, in the new release of the API, these parameters no longer exist and my implementation will need to be changed. The

[1]Sodalite is a royal blue mineral and stands for **S**tories for **O**n-boarding as **D**ocumentation **A**uthoring, **L**everaging **I**DEs for **T**ext **E**nahancements.

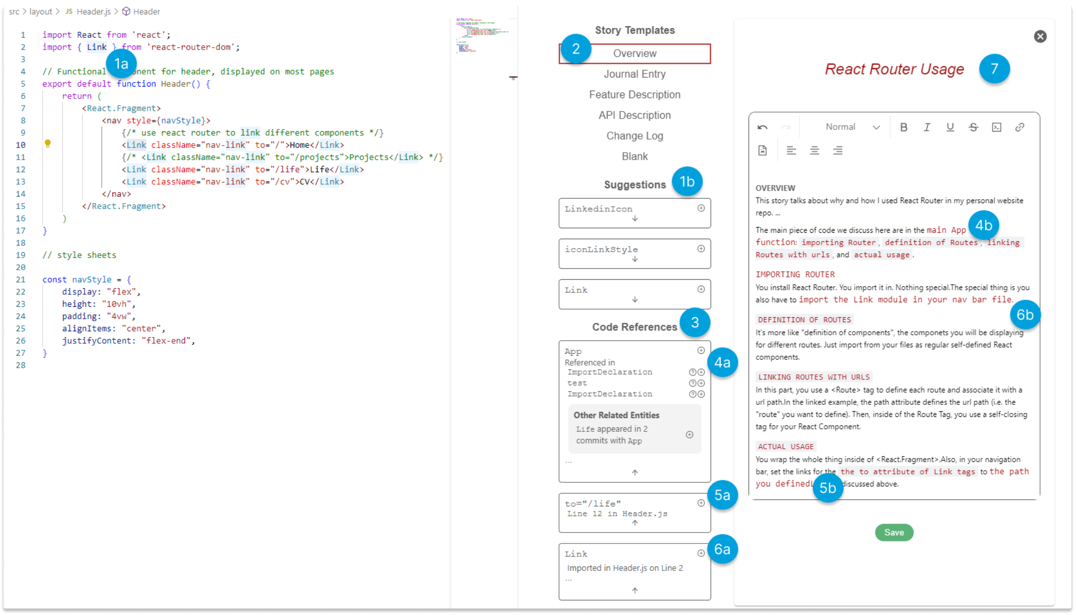
challenge of keeping code and documentation in sync with one another has been a long-standing research problem [4, 137, 163].

This chapter explores utilizing the connection between code and text to overcome longstanding barriers in documentation authoring and maintenance. By leveraging code meta-information including the structure, existence, and location of code, Sodalite is able to automatically determine the likelihood of the documentation being out-of-date given how well the system performs at re-attaching each code-text pair to the code in the IDE. Sodalite also uses this meta-information to help the initial documentation author in writing their "code story" through suggesting code patches to document, given the author's already-documented code, and providing templates for bootstrapping the authoring process. We expect this documentation authoring and maintaining process to be particularly useful for developer teams or open source projects, where the documentation and corresponding source code can "live together", with team-oriented documentation suffering from many of the same barriers that end-user facing documentation does [74, 135], including out-of-date information [135], while typically lacking a dedicated team or process for updating the documentation [3, 38].

5.2 Background and Related Work

Prior work about writing software documentation have ranged from understanding what information is in documentation [61, 103], what the problems are with that information [3, 4, 26, 45, 93, 108, 118, 137, 139, 163], how software documentation is authored [38, 143, 152], and automating documentation processes to offset authoring costs and standardize the information present in documentation [2, 55]. Notably, the majority of this documentation work is in the context of software documentation for end users of a software library, e.g. API documentation [115]. Most relevant to our work are studies about documentation created for other developers working on the *same code base*, such as internal documentation about the code base [135, 143] or a code tour for an open source repository [158], where the primary goal is to help other developers understand and contribute to the code base. Our work expands upon this work by contributing a system designed specifically to help create this type of documentation, with a focus on combating some of the known issues in authoring and maintaining this documentation including out-of-date information.

Once some software documentation is made, researchers have studied how developers maintain those documents and use that information. In studies of usage, researchers have identified many problems of documentation that lead to the documentation being less trustworthy [100], including questions about how up-to-date the information is [4, 93, 163] and completeness [137, 139]. A survey of developers at one company reported that they rarely updated documentation and that they correctly assumed that documentation content is out-of-date [93]. One reason for this lack of maintenance is that finding the appropriate places to update given a change can be challenging, with developers reporting that they would value a tool that helps identify those locations [50]. Our system attempts to reduce some of these costs by having maintenance be a core design consideration by automatically locating and highlighting the out-of-date portions of the document given which code was changed.



```
import React from 'react';
import { Link } from 'react-router-dom';

// Functional [illegible]nent for header, displayed on most pages
export default function Header() {
    return (
        <React.Fragment>
            <nav style={navStyle}>
                {/* use react router to link different components */}
                <Link className="nav-link" to="/">Home</Link>
                {/* <Link className="nav-link" to="/projects">Projects</Link> */}
                <Link className="nav-link" to="/life">Life</Link>
                <Link className="nav-link" to="/cv">CV</Link>
            </nav>
        </React.Fragment>
    )
}

// style sheets

const navStyle = {
    display: "flex",
    height: "10vh",
    padding: "4vw",
    alignItems: "center",
    justifyContent: "flex-end",
}
```

App
Referenced in
ImportDeclaration
test
ImportDeclaration

Other Related Entities
Life appeared in 2 commits with App
...

to="/life"
Line 12 in Header.js

Link
Imported in Header.js on Line 2
...

React Router Usage

OVERVIEW
This story talks about why and how I used React Router in my personal website repo. ...

The main piece of code we discuss here are in the main App function: importing Router, definition of Routes, linking Routes with urls, and actual usage.

IMPORTING ROUTER
You install React Router. You import it in. Nothing special.The special thing is you also have to import the Link module in your nav bar file.

DEFINITION OF ROUTES
It's more like "definition of components", the componets you will be displaying for different routes. Just import from your files as regular self-defined React components.

LINKING ROUTES WITH URLS
In this part, you use a <Route> tag to define each route and associate it with a url path.In the linked example, the path attribute defines the url path (i.e. the "route" you want to define). Then, inside of the Route Tag, you use a self-closing tag for your React Component.

ACTUAL USAGE
You wrap the whole thing inside of <React.Fragment>.Also, in your navigation bar, set the links for the the to attribute of Link tags to the path you definedL[illegible] discussed above.

FIGURE 5.1: The editor for authoring a story using Sodalite – this is a simplified and anonymous recreation of P2's story for demonstrative purposes. (1a) and (1b) show the relationship between the code editor and the story editor – in this case, the user has clicked the code `Link`, prompting the suggestions pane to show identifiers related to `Link`. (2) is the template list with the selected "Overview" template highlighted. The "Code References" at (3) are represented in a list (4a, 5a, and 6a) and their locations are shown in the story text at (4b), (5b), and (6b), respectively. The title of the story is (7).

5.3 Sodalite

In order to use Sodalite, a user begins by opening the Sodalite webview [111] which appears and functions like any other file in the editor (i.e., it can be dragged, resized, closed, etc.). The user can then view the stories that have been associated with the user's currently-open GitHub repository or choose to author a new story (see Figure 5.1). When authoring a new story, the user will be presented with a rich text editor, hereafter referred to as the "story editor" (in contrast to the "code editor", which refers to the Visual Studio Code IDE), and users can utilize code story *templates* and *code links*. For already-authored code stories, the system performs its maintenance algorithm to determine what code links are potentially invalid.

5.3.1 Templates

We included templates for stories to help guide authors about what should be documented (a common problem with documentation authoring is not knowing what is important to document [38]) and to provide built-in mechanisms for allowing developers to find the right code and code-related information to include in the documentation.

We began by identifying various types of developer long-form writings that are not adequately supported by current tooling and would benefit from leveraging the context of the source code. We compiled this list through a combination of reviewing literature and informally consulting with software engineers. For each of these types, we developed a "template" which contains a list of headings, subheadings, and guided instructions with prioritized code links (see Section 5.3.2), that would most likely be included. This list includes:

- **Overview documents**, which serve to introduce a newcomer to a code base [135, 158]. This template prioritizes higher-level information, including references to functions and classes, with the assumption that developers most likely do not want to cover lower-level specifics in an overview. "Overview" also prioritizes identifiers with certain keywords in their names that are commonly associated with control flow in a web development project, such as "listen" or "handle", with the rationale that the logical flow of a project may be discussed in an overview.

- **Journal entries**, which serve as logs about what a software engineer completed in a workday and are sometimes required by management [102].

- **Feature descriptions**, in which a developer discusses what a particular part of a code base does and is has been claimed to be one of the most important types of documentation for developers [93].

- **API descriptions**, where a developer catalogs what they have learned about an API to benefit later developers [65].

- **Change logs**, in which a developer enumerates parts of the code base that have changed between releases of a software project [102], but, due to authoring costs, can be incomplete [28].

Note that we are not claiming that this is an exhaustive list and our system allows developers to define their own template if the built-in ones are not adequate by using a predefined JavaScript object notation (JSON) structure that Sodalite reads. The system also defaults to a "Blank" template, such that the user can define their own structure without extending Sodalite.

5.3.2 Code Links and Suggestions

A key feature of Sodalite are *code links*, where the story references back to code. Links can be made manually or using the "Suggestions" pane (see Figure 5.1-1b), which lists code links that the author may include in their code story. Code links come in three varieties:

- **Identifier definitions**, which link to where a specific code entity like a method or variable is first defined. A user can create an identifier definition link by selecting the "plus" button in the suggestions pane at the top right corner of the code link box. Identifier definitions also include additional information about the identifier, including places in which it is referenced, and other classes or functions it references (see Figure 5.1-4a).

- **Identifier references**, which link to a specific instance in which a particular identifier is referenced or used. An author can make an identifier reference by selecting a reference that is listed in an identifier definition's list of referenced locations. Figure 5.1-5a shows a reference.

- **Code ranges** (see Figure 5.1-6a), which can be any arbitrary range of code that the user has selected in the Visual Studio Code editor. Selected code in the code editor will always appear at the top of the user's "Suggestions" list, given that selecting code is a strong signal that the user wants to include that code in their story. Code ranges are fundamentally the same as the "code anchors" that Catseye supports.

Code links can be added to a code story in two different ways. If the user has no text selected in the story editor, the code link will either insert the name of the identifier (identifier definitions and references) or the selected code (code range) at the location of the user's cursor. If the user has selected some text in their code story, that text will be linked to their code (see Figure 5.1-4b, 5b and 6b). Either way, clicking on a code link within the story editor will navigate to wherever that particular code link is located in the code editor. Once a code link has been added to a story, it will appear in a "Code References" list (Figure 5.1-3), such that the link may be used elsewhere in the story.

Code links can also contain additional meta-data Sodalite was able to determine about that part of the code. This includes, for the identifiers, where they are defined, and referenced in different parts of the code. Once a code link has been included in the code story the "Suggestions" pane will include other identifiers that were commonly edited at the same time as that particular identifier. We identify these "co-edits" by parsing the Git commit history for that particular identifier and count when other identifiers appear in the same commit. In this way, we attempt to identify parts of code that are related but not in a way AST parsing would find.

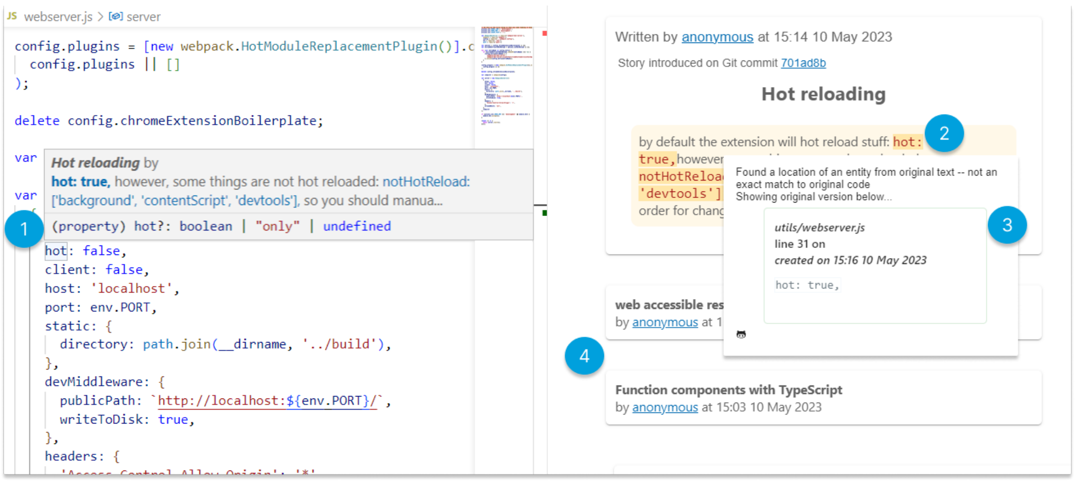
Given these different types of code links, the "Suggestions" pane uses different sources of information to populate its list. Sodalite examines both what the user is doing in the story editor and what they are doing in the code editor. The information the algorithm leverages from the story includes what references are already included in the story and what the user has most recently typed. Identifiers related to references already within the code story will be prioritized, as will identifiers that match some part of the most recently-typed text. The system then complements that information with what it knows about the user's current location within the code, including if the user is currently selecting an identifier, and, if so, what identifiers are related to the selected identifier, along with other identifiers referenced in the file and, if applicable, in the user's currently selected scope. Some templates like the "Change Log" prioritize certain types of code links (e.g., code links that have been heavily edited), in which case the system favors those identifiers in the Suggestions list. Identifiers that appear in multiple information sources are pushed to the top of the list. To prevent the user from being overwhelmed with suggestions, we limit the amount of suggestions to the top 10.

When a story is saved, Sodalite generates a JSON file in a system-generated folder, which can be committed to the user's Git repository and used by other programmers when they read or edit the source code. This JSON file is also used by the system when determining what code links are out-of-date.

5.3.3 Support for Reading

Sodalite has some features designed specifically to help readers of code stories better understand and utilize the documentation. A core feature of Sodalite for readers of code stories is the fact that it is situated within the context of the code. Developers have previously stated they value source code and code comments more than other types of documentation [148] – we hypothesize that bringing the type of information typically in external documentation into the source code will allow for "the best of both worlds" by staying within the developer's working context while also supporting longer-form text.

Another feature of Sodalite that we expect to help users of code stories are the bi-directional nature of the code links. When some code has been linked in a story, the code will be highlighted within the IDE, such that users of the code can discover



```
config.plugins = [new webpack.HotModuleReplacementPlugin()].c
  config.plugins || []
);

delete config.chromeExtensionBoilerplate;

var
var
```

Hot reloading by
hot: true, however, some things are not hot reloaded: notHotReload: ['background', 'contentScript', 'devtools'], so you should manua...

```
(property) hot?: boolean | "only" | undefined
    hot: false,
    client: false,
    host: 'localhost',
    port: env.PORT,
    static: {
      directory: path.join(__dirname, '../build'),
    },
    devMiddleware: {
      publicPath: `http://localhost:${env.PORT}/`,
      writeToDisk: true,
    },
    headers: {
```

Written by anonymous at 15:14 10 May 2023

Story introduced on Git commit 701ad8b

Hot reloading

by default the extension will hot reload stuff: `hot: true,` however `notHotReloa` `'devtools']` order for chang

Found a location of an entity from original text -- not an exact match to original code
Showing original version below...

utils/webserver.js
line 31 on
created on 15:16 10 May 2023

```
hot: true,
```

web accessible res
by anonymous at 1

Function components with TypeScript
by anonymous at 15:03 10 May 2023

FIGURE 5.2: How Sodalite appears after a story has been authored. (1) shows the hover text when a code link is interacted with in the code editor. (2) is a code link that has been marked as "needs review" given that the original code (shown in (3)) and the code in the editor (at (1)) are different. (4) are two other collapsed code stories.

pertinent documentation that is relevant to that code (see Figure 5.2-1). The highlighted code brings up a hover text that shows a preview of the code story, including the surrounding text from the story, the part of the story that is linked to the code in bold, and the name and author of the story. If the surrounding story text contains code links to other parts of the code, those links will navigate the user to the code link's location. If the user clicks on the link in the hover text, the Sodalite pane will open and scroll to the correct part of the story. This does not interfere with the regular links in Visual Studio Code's hover text, which still work like normal.

Users of code stories are also presented with additional context about the originally authored story, to further help them validate what information is still relevant. The story shows who authored the story, the date and what Git version the project was on when the story was last authored or modified, and each code link in the story displays additional metadata about what version of the code it was on when authored, along with its original code content and context. This metadata complements the information the out-of-date system provides when the story is validated.

5.3.4 Support for Maintenance

Sodalite leverages being in the code editor and having access to Git versioning information to be able to mark parts of stories as "valid" (in which all code links are valid), potentially in"need of review" (in which the system found a potential match for the code link, but it is not positive – see Figure 5.2) or definitely "invalid" based on how well the system is able to match the code references in the story to the code currently in the user's project in Visual Studio Code. There are situations in which the documentation may go out-of-date that our system would not capture, but, in a study of documentation problems, [4] found that most cases in which the documentation went out-of-date was due to the code changing, so we focus on that case with Sodalite.

On launch, Sodalite parses every code story file in the user's current project and builds an internal AST representation of the code in the project to compare the code links against. We use the different types of code links to inform how to re-attach a particular link and whether that attachment is valid or not.

For code anchors, given their flexible code structures (e.g., anything from a string to a full multi-line expression), we begin by evaluating the most optimistic condition, in which nothing has changed, through checking whether the position we have saved contains the same code as the code link. If not, we then look for whether the code from the code link exists anywhere within the document. If that does not work, we use purely text-based matching mechanisms to discern candidate matching points, since that was the most successful method used in [134]. We use AST information to restrict our search spaces, using a recursive function from most within the last-known scope location.

In the case of multiple matches or no matches, we run our re-anchoring algorithm, partially adapted from [134]. This algorithm uses the last-known code content, the original location within the file, and the original surrounding code (all of which are saved with each code link) to find the most likely candidate code reference. We then weigh the probability that the location is correct through a combination of calculating the edit-distance between the two versions, the edit-distance between the surrounding lines of code, and the difference between the candidate line(s) of code and the original location, with a closer location being weighted higher.

If there is a low score or no matches, we mark the anchor as invalid and the corresponding text in the story as in need of review. To assist the author in finding either a new location to link the story text to in the code or to discard the link, we present additional metadata about what the code looked like and where it was last located (see Figure 5.2-3, where it says "Showing original version below...").

For definition references, the system searches the internal representation to see whether the identifier is defined at its last known location. If the definition is not there, the search will expand outwards to see if the identifier is defined in a different file and, if so, attaches to that location, but marks the link as potentially invalid, considering it may be a different identifier that just happens to have the same name. If it is a local definition (e.g., a property named `bar` that is part of class `Foo`) the system will check that both the class `Foo` and the property `bar` exist. If the system cannot find a definition for the particular referenced code entity, the system uses the text-and-location based re-anchoring algorithm. If the algorithm does not return a result of a sufficiently highly-weighted likelihood, we mark the reference as invalid, and the surrounding text within the code story as in need of review by the author.

A similar approach is used for checking identifier references in the code. We begin by seeing whether there are one or more references in the code in the last-known scope in which the reference was used. If there are multiple candidate matches, we find the reference with the most similar AST path to the saved code anchor information. If there are no candidate matches, we once again fall back on the text-matching algorithm and, if the result is inadequate, mark the text within the story as potentially invalid and in need of review.

By attempting to automate the process of re-matching the story content to the corresponding code and suggesting potential edits to make, we attempt to lower the workload for the author in maintaining the documentation. "Valid" code links have no highlights. "In need-of-review" code links (shown in Figure 5.2-2) are highlighted in yellow as a warning to readers. "Invalid" code links are highlighted in red to draw attention to this part of the story.

5.4 Evaluation of Sodalite

5.4.1 Study Design

In order to assess the usability and usefulness of Sodalite, we ran a small user study to test both the authoring and maintaining aspects of Sodalite. Participants were instructed to use Sodalite to document some code of their choosing. We then observed them as they chose to document whatever information they saw fit.

To evaluate the maintenance features, participants took their documented code and reverted to an *older* version of their code. We considered requiring participants to document an old version of their code, but decided against this since we did not want the study to become a test of how well the participant's remembered their old code and developers are not typically documenting their old code. Therefore, they documented the current version, and we pretended that an old version was a newer version.

Participants were instructed to look through their project's GitHub history to find a version that was many commits behind the version they documented and involved edits to any files that they ended up documenting during the authoring portion of the study. Upon finding an appropriate commit, participants checked out that version of the code and re-ran Sodalite, which triggers the maintenance system to try and find appropriate re-anchoring points. The first author and the participant stepped through each code link within the stories to determine whether the system chose the "correct" action, meaning the participant agreed with the system's determination of whether the link was "valid", in "need of review" or "invalid".

We recruited 4 participants (3 men and 1 woman), hereafter referred to as P1 through P4, using study recruitment channels at our institution and advertisements on Twitter. All participants were required to have some amount of experience using JavaScript or TypeScript, to have a project written in one of those programming languages that they were willing to document, and to regularly use Visual Studio Code. All sessions were conducted over video conferencing software and the sessions were recorded. Participants had, on average, 12.25 years of professional programming experience and considered themselves proficient in JavaScript and TypeScript.

For analyzing the authoring experience, we captured how many stories the participants authored, how long each story was, what type of templates the participants used, how many code links they chose to include across how many files, and what types of code links they were. We additionally reviewed the recorded videos to count whenever the participant navigated through the document using the code links, and counted whenever a participant copied and pasted or deleted a code link. We additionally noted any usability issues participants discovered with the tool.

To assess how well Sodalite did at *maintaining* the code stories, we computed the false positives, false negatives, true positives and true negatives for each code link given the participant's evaluation of whether or not Sodalite correctly re-attached the code link. We additionally logged what the offset was from the code location during the authoring session to the maintenance session, along with the total diff between versions, and what type of re-attachment strategy the system used (e.g., text-based, AST based, etc.).

5.4.2 Study Results

All participants were able to successfully use Sodalite to author at least one code story that utilized Sodalite's features, and agreed with Sodalite's re-attachment decisions, on average, 86.5% of the time. We further discuss participants' experience using Sodalite's authoring and maintenance features.

Authoring

The 4 participants authored, in total, 15 stories, with 2 participants each writing one story, and the other 2 participants authoring 6 and 7 stories, respectively. On average, participant's wrote 238 words (min = 136, max = 312, std. dev. = 75.4) during the 45 minute authoring time. The participants in our study chose code repositories of varying purposes, sizes and complexity to document. Participants, on average, chose to document 3.25 files. Our participants chose to document a Google Chrome extension, a Visual Studio Code extension, a browser game, and a personal website.

Across all stories, participants created 52 code links, with each participant, on average, creating 13 (min = 11, max = 18, std. dev. = 3.36). 18 of the code links were pointers to definitions of identifiers within their code[2], 7 were identifier references, and the remaining 27 were "code ranges". In their stories, participants, on average created 12.75 code links.

The code links were represented in different ways within the code story. 29 of the 51 code links were attached to text in the code story that was just the name of the identifier or the expression the user had selected. Notably, 18 of those 29 code links were definitions of particular identifiers, where the story sought to explain some detail about that identifier. The remaining 22 code links were attached to text that was part of a sentence within their code story. This split in usages suggests that supporting both of these ways of referring to code helped support different ways in which participants wanted to talk about their code.

Participants sometimes used their code links to help them navigate through their code, using, on average, 41% of their code links at least once to navigate through their code (min = 0, max = .722). Participants would sometimes check, after including a code link, where that link went as a way of assessing the reader experience. P4, in particular, used the navigation features often, revisiting the majority of their code links to review the other places in which the code was referenced and used their links to navigate 13 different times across the session. This suggests that the code links may be useful for navigating through a code base, even when authoring documentation.

Maintaining

Sodalite made the correct decision of either re-attaching or not re-attaching 44 out of the 52 code links, a success rate of 86.5%. Of the incorrect links, 7 were because the system could not find an attachment point, despite there being one within the code, and 1 was because the system erroneously found a location that it thought was correct when it was not. The majority of cases were that Sodalite successfully re-attached a code link, at 36 instances. Eight code links had changed in such a way the system correctly marked them as needing review. The reversions averaged 100 lines of code lost (min = 53, max = 137, std. dev. = 38.92) and 49 lines of code added (min = 3, max = 95, std. dev = 51.97) per file.

[2]P1 made exclusively this type of code link, comprising 11 of the 18 total definitions.

Sixteen of the correctly re-attached links were definitions, in which the re-attachment strategy was to find a point within the file in which the named entity was defined. Given the relative broadness of this strategy and the specificity of the named functions or classes participants wanted to document, all but 2 definitions were able to find an appropriate point somewhere in the document. This strategy may not work as well if a method or class is renamed, but that did not happen to occur in any of the participants' versions. The reference re-attachment strategy also worked well, with 8 of the 10 references being successfully re-attached in an appropriate location.

Perhaps more interestingly, the 8 cases that the system correctly marked for review occurred in different situations, including when some code had been commented out, the underlying semantics of the code changed significantly enough such that the text in the story was no longer valid, and when the code links' corresponding definitions did not exist. One such instance is shown in Figure 5.2 in which the original code link and surrounding text states that a value, `hot`, should be set to true. However, the code in the version the user has opened has `hot` set to false. Two of the cases occurred because the definitions for the linked methods did not exist in that version of the code.

The single false negative case, where Sodalite missed a change, also occurred using the text-based matching algorithm. The anchor was originally placed on an `else` keyword, but, when the prior version was pulled in, the system placed the code anchor on a completely different instance of `else`. This other location was closer to where the original link was and, since `else` is such a common keyword, the system found multiple reasonable locations but selected the incorrect one. For identifier references, Sodalite flags situations like this, but the system does not currently support that for keywords. Future versions of Sodalite would benefit from introducing more heuristics about the context in which the code appears, including any logical dependencies, such as an `else` statement's corresponding `if` statement.

5.5 Discussion and Future Work

Our evaluation lends evidence to the claim that in-editor authoring and maintaining of documentation can be supported with Sodalite. Participants succeeded in creating code stories and, when faced with changed code, the system was very successful in identifying problematic references, and relatively successful in choosing the appropriate action to take.

A current limitation of Sodalite is that all of the mechanisms for determining whether or not the story is out-of-date are contingent upon the story including code links. The expectation is that, given Sodalite being located within the IDE, developers will naturally utilize that context and reference their code within their code stories. Indeed, all participants stated in the post-task survey that they valued the ability to link their text to their code. Nonetheless, future versions of Sodalite may benefit from additional mechanisms for assessing the validity of the text in comparison to the code, for example, through leveraging crowd-sourcing mechanisms for quality control [6, 38].

Another benefit of having even better mechanisms for identifying out-of-date code stories is to notify documentation writers and/or developers when their story has gone out-of-date. It is a known problem that updating documentation is a task that developers typically put off – while Sodalite attempts to make updating easier through marking places for review, the system does not require that the author take any action. One can imagine an additional feature to Sodalite that allows stories to be marked as “very important”, in which case developers could immediately be notified if they make a breaking change, while lower priority stories can be addressed on the developer’s own time. In the proposed work which involves integrating Sodalite with the Meta-Manager, Meta-Manager’s constant listening for change events may be leveraged for real-time detection of code links going out-of-date.

Chapter 6

Meta-Manager: Meta-Information for Question-Answering

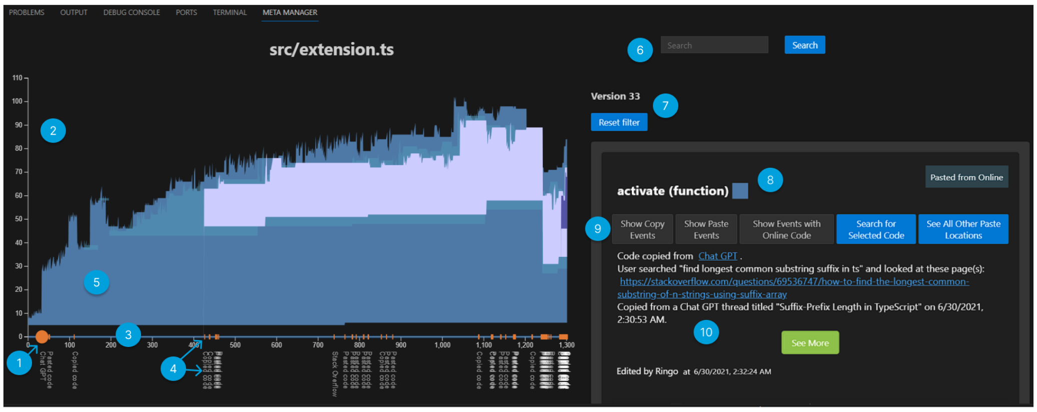
This chapter is adapted from my paper:

[66] Amber Horvath, Andrew Macvean, and Brad A. Myers. 2024. "Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code". *Currently in-submission*.

6.1 Overview

When developing and understanding code, as previously discussed, developers are managing many types of information that come in different forms. The previous chapters explore how this information can be externalized and made more useful through tooling approaches that leverage the context of the source code. However, this externalization requires additional cognitive and physical effort by the initial author [12, 25, 62, 80, 141], which can be a deterrent from using these types of tools, and only capture what is expressly written by the developer, which, while useful for capturing information that may exist only within the head of the developer, can miss other interesting or useful contextual meta-information that was created during the sensemaking [98, 99]. Nonetheless, information related to the history and implementation of code, while often not expressed by the developer [94], is often useful in answering historically "hard-to-answer" questions about code, specifically those related to rationale [88, 91].

In this chapter, we explore capturing code editing and history information traces at scale through an automatic, event-driven tooling approach. We isolated editing events that we expected would help developers in answering some of these questions, such as copy-paste events to help with understanding *where* some code originated from and web search events and additional meta-information from web pages including Stack Overflow and ChatGPT for answering *why* some code is written a certain way. We instantiated this event-driven code provenance model in our tool, Meta-Manager. Meta-Manager translates this information into key events along a code history timeline visualization that allows developers to explore how, when, and where some code came to be. This visualization and accompanying details panels support filtering and searching to help developers isolate and investigate the information relevant to the specific parts of the code that they are investigating. We found that users could use its code history exploration support for answering otherwise impossible-to-answer questions about code.



Code copied from Chat GPT .
User searched "find longest common substring suffix in ts" and looked at these page(s): https://stackoverflow.com/questions/69536747/how-to-find-the-longest-common-substring-of-n-strings-using-suffix-array
Copied from a Chat GPT thread titled "Suffix-Prefix Length in TypeScript" on 6/30/2021, 2:30:53 AM.

Edited by Ringo at 6/30/2021, 2:32:24 AM

FIGURE 6.1: Meta-Manager as it appears within Visual Studio Code: the pane appears in the bottom area of the editor, with the left area displaying a visualization of the history of the code file over time, while the right area displays information about a particular code version.

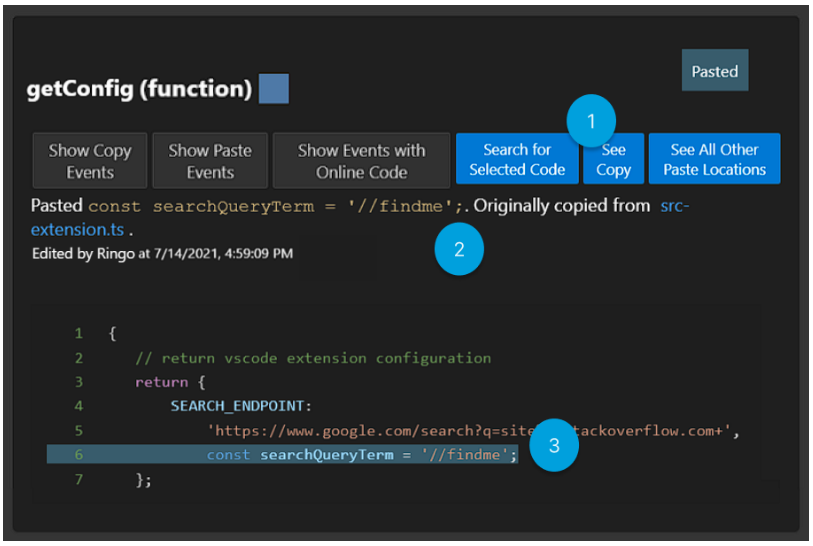
6.2 Meta-Manager

We developed Meta-Manager, a Visual Studio Code extension with an accompanying Chrome browser extension, that organizes and keeps track of code editing history, with a particular focus on extracting interesting events in the code history (see Figure 6.1). In order for Meta-Manager to begin logging code versions, the user does not need to take any actions beyond installing the extensions. On each file save, Meta-Manager will log a new version and perform an audit of the file to see if there are any new blocks of code to track. To investigate the code history, the developer can navigate to the "Meta Manager" tab in the bottom area of the editor — doing so will render the edit history of the user's currently-open file. Whenever the user opens a file, the Meta-Manager will render that particular file's history.

With the Meta-Manager pane open, a developer can explore their code's history in a variety of ways. The default view of the code history, as shown in Figure 6.1, will annotate the code timeline with a variety of event labels, which correspond to particular code versions where something that might be of interest happened. To navigate to a particular code version, the developer can either click on the event label (Figure 6.1-4) or use the scrubber (Figure 6.1-1) to drag the scrubber to the version of interest along the x-axis. While scrubbing, the right side of the Meta-Manager view will update with information about that particular code version, including what the code looked like at that point in time.

With a search box and/or filters, the developer can find other related events and search based upon the content within the code version (Figure 6.1-6, 9). With the search box, developers can search backwards through time for any text that ever appeared in the code. The filter buttons operate on the current block or function (which is the `activate` function in Figure 6.1-9). "Show Copy Events" will filter the timeline to show all points at which code within that block was copied. "Show Paste Events" will, similarly, filter the timeline to show all points at which some code was pasted within the block. "Show Events with Online Code" will filter to show only instances in which that block of code had some code pasted from an online source.

Once the user has found a code version of interest, they can explore its history. The history view on the right displays a list of code boxes, where each box corresponds to a blocks of code within the file. These blocks then correspond to the



Pasted `const searchQueryTerm = '//findme';`. Originally copied from src-extension.ts .
Edited by Ringo at 7/14/2021, 4:59:09 PM

```
1  {
2      // return vscode extension configuration
3      return {
4          SEARCH_ENDPOINT:
5              'https://www.google.com/search?q=site    ackoverflow.com+',
6              const searchQueryTerm = '//findme';
7      };
```

FIGURE 6.2: How the code box looks when expanded to show a code version – in this case, a "Paste" event version.

different colored parts in the visualization – in Figure 6.2, the large blue area in the chart corresponds to the growing, shrinking, and moving of the `activate` function. In the case of nested blocks (e.g., a method within a class), the boxes will be indented and the colors in the visualization will overlap, such as the violet area on the chart covering the dark blue. Each code box (Figure 6.2-8) will display meta-information about that particular code version, including when the code was edited, who it was edited by, and what the code looked like at that point in time (see also Figure 6.2-3).

A key design feature of Meta-Manager is tracking events that happen to the code over time. For certain events, additional meta-information will be shown on those particular code versions. For example, in Figure 6.2, this particular version of the method `getConfig` had a paste event, where the user pasted in the code `const searchQueryTerm = '//findme';` on line 6. The version adds additional information such as where that copy came from (in this case, the file "src-extension.ts"). Clicking the "See Copy" button (Figure 6.2-1) will either filter the timeline scrubber to show the point in time at which the copy occurred if the copy happened in the same file, or will display a preview of how the copied code looked at the time of the copy if the code that was copied came from a different file[1]. "See All Other Paste Locations" will similarly filter the timeline to show other points at which that same code was pasted, and/or show previews of other locations in which the paste appeared outside of the current file.

In cases where code was pasted from an online source, Meta-Manager will provide additional meta information about the web page that the code was pasted from, and, if available, what the original user was attempting to do. Meta-Manager's supplementary browser extension is designed to work with some popular programming learning resources, including Stack Overflow, GitHub, and ChatGPT[2]. If the browser extension detects that the user is on one of these web pages, it will extract website-specific information (e.g., the thread name of a ChatGPT thread, or a Stack Overflow

[1]We show a preview rather than navigating a user out of their current file and changing the whole timeline for the other file, since this would cause the user to lose their current context. Clicking on the preview can cause this navigation to happen if the user wants.

[2]We envision this list being substantially expanded to include other commonly-used resources where code is copied from, such as the official documentation for languages and APIs.

question and vote count) and listen for copy events. If the Visual Studio Code extension detects a paste which matches the content of the browser extension's copy, this additional information will be transmitted to the Visual Studio Code extension to be associated with that paste. The hypothesis is that the query text can be a good signal of the developer's original intent for the code, which has been supported by prior work [80, 100] and our observations.

Similarly, if the user makes a programming-related Google search[3] prior to visiting these websites, their initial query and visited web pages will be included with the meta information about the pasted code. This information about the user's web browsing history and eventual copied code can then be seen with the paste version (Figure 6.1-10). Clicking the "See More" button will pull up a preview of the web page in the Meta-Manager pane of the editor, and highlight the part of the code on the web page from where it was copied.

In order to help developers navigate the timeline and find specific versions of interest, users can search through the code versions. Users can search across time using either code that they have selected in their current code version (Figure 6.2-1, "Search for Selected Code") or directly through the code editor by selecting some code in their file, then using the context menu to select "Meta Manager: Search for Code Across Time". These two searches differ slightly from one another, in that the search using the code box will search *forwards* in time from the specific code version, while the search from the code editor will search *backwards* in time (since the editor code view always shows the current version of the code). Both searches utilize the edit history by modifying the query given how the code changes across each version. This means that the search will attempt to expand if the selection grows, shrink if the selection shrinks, and update the code query content to match on given variable names and other constructs changing over time. For example, if a user searches backwards in time on the code `const match = 10` and Meta-Manager detects a change in that expression, the query will update to search on the new expression e.g., `const match = 12` such that, as we continue moving backwards through time, we will be looking for a more accurate representation of that code at that time.

When a search is performed, the timeline will update with events marked "Search Result" for events affecting the specified code, where the code differs in some way from the previous version. This is to prevent the search results from being flooded with events where the code is exactly the same, but has either moved as a result of other code above it being edited, or not changed at all. When looking at a search result code version, the part of the code that matched the user's query will be highlighted in orange. The search will also look out for significant edits made to the code. This includes when the searched-upon code is initially added, removed, commented out, or commented back in. These events are specifically marked on the timeline with a label corresponding to the type of edit the searched-on code experienced (e.g., "Commented Out").

Since the sheer amount of code versions will accrue over time, Meta-Manager allows developers to zoom in to parts of the visualization that they find particularly interesting or have a large number of relevant events in a short amount of time. Users can do this by setting the scrubber at the start of the edit history they want to zoom in to, then hold the shift key and click to the end of the zoomed portion. The

[3]We consider a programming-related Google search to be one in which popular programming-related websites appear in the search result list. We acknowledge the potential privacy problems with this feature, and consider the current prototype to mainly be an evaluation of the advantages of doing this, and expect that a more complete tool can provide more control over what is saved from the browser, as in [100].

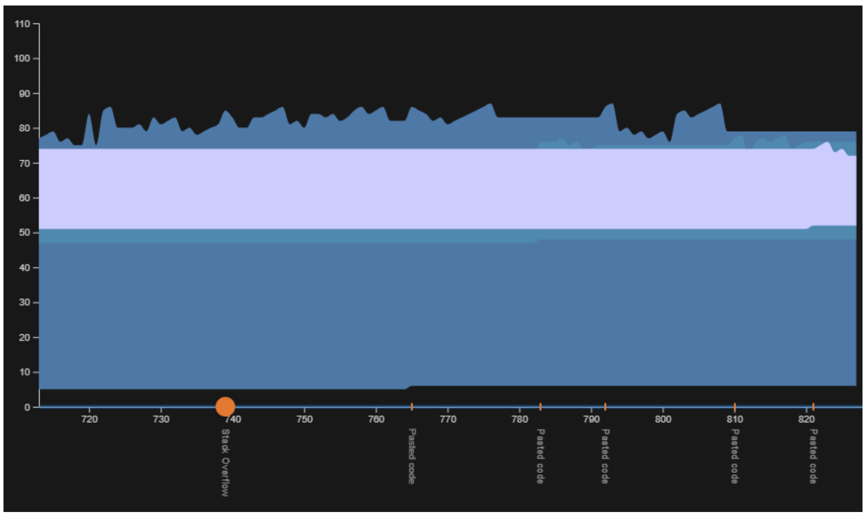


FIGURE 6.3: A zoomed-in portion of the timeline shown in Figure 6.1. This zoomed-in portion shows around 120 edits between Version 710 and Version 830, with the scrubber set around Version 740, when a user pasted code from Stack Overflow.

visualization will update to show a slice of the editing history (Figure 6.3), which can be eventually dismissed with a "Reset" button.

6.2.1 Background and Design Goals

In designing Meta-Manager, we aimed to collect the most amount of potentially meaningful information with the least amount of effort required by the developer, and then present that information in an organized and explorable manner. Considering how cognitively demanding programming itself is, we wanted the interactions with Meta-Manager to be familiar (i.e., support traditional searching and filtering mechanisms) while pulling out particular events of interest that may indicate more important changes happening to the code. To that end, we chose to focus on copying code, pasting code, commenting in and commenting out code, and introducing or removing code that the user has shown interest in. We chose these particular activities because of how they relate to some of the previously-reported "hard-to-answer" questions about code; we envision logging these activities may help with answering the following questions:

- **How has this code changed over time?** [91, 146] Developers often try and understand the evolution of some code in service of answering a question that is pertinent to their current task. For example, this may help while investigating when a bug was introduced [147], finding when some code was last used in service of understanding how a feature changed over time [171], or finding a snippet of code that was edited repeatedly to understand where the original developer had issues [91, 94, 147]. Isolating when these particular changes happened can be impossible in the case that the intermittent version is not logged in a version control system (which is often the case in situations where a developer is trying out multiple solutions), or very difficult to find even if there. We address this challenge through making the code history viewable, interactive with a scrubbing functionality, and searchable, with situations such as code being removed or commented out explicitly called out in the version history's timeline.

- **Why was this code written this way?** [83, 91, 146] A commonly-reported activity among developers when understanding unfamiliar code is reasoning about why it is written the way that it is. This information is typically only known by the original author during the time at which the code was written and, if not written down, is usually lost. On the off chance it is recorded, it is most likely preserved in the form of a random Git commit message or code review comment, which are often too difficult to forage through [149]. When the code of interest results from a paste or AI auto-complete event (e.g., from CoPilot), then we posit that, by providing contextual clues about the original code author's intent through recording their web activities and connecting these activities with the relevant code edits, later developers can use this information to better reason about what the original developer was trying to achieve. Literature has reported that developers issue, on average, more than 20 search queries related to their programming every day – capturing this information and associating these queries with their related edits may serve as useful information for later developers [11, 170]. In cases where the code of interest is not related to an event Meta-Manager currently logs, we envision that these "why" questions may be supported through other contextual cues, which we discuss in Section 6.4.

- **What code is related to this code?** [36, 91, 146] Oftentimes, when contributing a change to a code base, developers must reason about how their new code is related to many other parts of the code beyond simply what could be found in a call graph. Other relationships that developers reason about are what parts of the code are commonly edited together (e.g.., the "working set" of code patches that makes up a particular feature [18, 32]), and, if introducing a change or refactoring some code, what other parts of the code must be updated. If code was copy and pasted around, that is a strong signal that the copy-paste patches are related, so we posit that investigating those versions may help with understanding otherwise invisible relationships between code patches and their pasted "descendents". We further hypothesize that this can help with determining code locations to update when introducing a change, determining what code served as a template for some other code, and, perhaps, finding what parts of the code base need to be refactored in order to reduce redundancy [79]. Similarly, even if copy-and-paste are not used, code which is edited at the same time can be investigated by scrubbing along the timeline around edits of the current code[4].

- **Where did this code come from?** [146] In 2021, Stack Overflow reported that one out of every four users who visit a Stack Overflow question copy some code within five minutes of hitting the page, which totals over 40 million copies across over 7 million posts in the span of only two weeks [130]. Given this ubiquity of online code and developers reliance upon it, researchers have investigated the trustworthiness of code that is sourced from online resources [10], and, with the rise in large language models (LLMs) for code generation, research is beginning to focus on the quality of AI-generated code as well [78, 95, 97]. Typically, it is not easy to see what code came from AI or from an online source, versus what was written by developers themselves. While developers occasionally add code comments that cite where some pasted code came from,

[4]In the current prototype, this only works for edits in the same file — we leave such cross file investigations for future work.

this does not happen very often [9] and, when it does happen, the links have a tendency to break over time and, going through the effort of recreating the context in which that code was initially added and determining whether it is still valid is laborious [57]. Our system keeps a copy of the web page content that is relevant to the copy-paste event which is stable even if the link becomes no longer valid, isolates the part of the page that is relevant to the copy-pasted code, and presents this information in context, such that there is no costly context switching required by the developer.

It is also worth noting that all of these questions are phrased about "this code", which is the language used in the original research. This phrasing suggests that developers are typically discussing these questions at the snippet or block level [63], as opposed to the file level, which is what typical version control systems operate at. Our system differs by tracking code at the block level across files and supports drilling down to a specific line or lines of code interactively.

6.2.2 Implementation

The Meta-Manager, both the Visual Studio Code editor extension and supplementary browser extension, utilize TypeScript for the logic and React [46] (with D3.js [119] for the chart in the Visual Studio Code extension) for the front end. FireStore [41] is used for authenticating the user, establishing a shared connection between the browser extension and Visual Studio Code extension, and logging the code revisions and metadata in the Meta-Manager database.

The code logging in the editor works by utilizing the TypeScript abstract syntax tree (AST) on system launch to parse each file in the user's currently-open repository. The system then attempts to match each block within the parsed-AST to known code entities stored in the database. This matching uses a variety of heuristics including text-matching using the "bag of words" approach discussed in [168], the last-logged structure of the AST (such that known relationships between blocks are prioritized), the user's current Git commit and the logged versions' commits, and the line difference between blocks. In the case that a block in the history is not found in the current version, we consider it deleted, and, to cut down on superfluous versions, do not pull in its history[5]. If an unknown node is found in the current code AST, we assume it is newly created and begin tracking its history. We additionally listen for any document change events to capture whether or not a new block is added or removed to begin tracking or stop tracking the node's history. Each node within the system subsequently manages its own version history and the events that happen to it, such that users can drill down even further to see, e.g., when a particular code block was introduced or removed. While the user is editing their code, each node keeps what we call a "change buffer" of edits that happen to the node, with each edit parsed to determine whether it is an edit of interest, such as a paste event or a block code comment. In the case of a copy, the system identifies which node experienced the copy such that a connection between the respective versions can be made between that node and the node that receives the paste event, even though the copy event itself doesn't actually change the code.

[5]Note that, given the nested structure of the AST, if a missing node's parent is present in the user's code, the user can still see the missing code in the parent's version history and query upon it, if they want.

6.3 Lab Study

In order to assess how well Meta-Manager performs in helping developers answer hard-to-answer questions about code history, we ran a small user study. Participants were tasked with using Meta-Manager to explore an unfamiliar code base using the system to answer questions we asked them about the history of the code, without modifying or running the code. We chose to have a single condition (as opposed to a between or within subjects study design) in which participants used the tool due to the fact that the questions we asked participants would, without the tool, be unanswerable, meaning there is no real control condition we could grade the experimental condition against. This was done deliberately considering we specifically designed our tool to support answering these types of questions. Thus, ensuring that the tool succeeded in that regard was our primary goal of the study, along with assessing the usability and utility of the tool.

The lab study consisted of a tutorial with Meta-Manager in which the experimenter and participant walked through each feature. Then, the participant and experimenter walked through different parts of the code base and the participant would use Meta-Manager to try and answer each of 8 questions (Table 6.1). Once the participant answered each question, the study ended with a survey to capture participant demographic information, along with their experience using Meta-Manager, and their own history in attempting to answer the types of questions Meta-Manager is designed to help with answering.

6.3.1 Method

Code History Creation

Prior to running the study, I created a code base to perform the experiment on. Given that Meta-Manager has not existed long enough to naturally accrue a history log that would be in line with real, prolonged use of the tool, a code history was artificially created. We did this because we did not want to bias the study in favor of the tool purely because there are a small amount of code versions, thus finding an answer to a question is trivial. The artificial code base is based upon a real code base [150] for a Visual Studio Code extension created by an external group unaffiliated with Meta-Manager, which functions similarly to CoPilot. This repository was chosen due to the fact that much of the code centers around the Visual Studio Code API, which few developers are familiar with, thus lowering the likelihood of a participant performing well purely due to having more background knowledge in the domain.

To create the artificial edits, the first author independently rewrote the code base, following along with the Git commit history in order to capture "real" versions of the code. While writing the code present in each commit, the tool was logging these real versions, but was also recording individual edits (e.g., add 1 line that says `const searchResults = match(searchResults);` in file search.ts on commit 4acb) that were then artificially inserted at realistic intervals across each code's history, given the correct file, time period, and node. The first author intentionally did not write "perfect" code that matched what was in each commit, to account for the more realistic intermittent versions the tool would capture in real usage. The author also intentionally added events that we are particularly interested in investigating, such as copy-pastes, across each file's history, along with simulated copy-paste events that match the frequency reported in prior literature on how often developers copy-paste during a normal programming session [70]. We also added realistic copy and pastes from Stack Overflow and a few from ChatGPT (even though the code was actually written before ChatGPT was available) since these will be increasingly important, with these events occurring less frequently than within-editor copy-pastes. Using this technique, we generated a code base consisting of 5,661 edits in 1,328 lines of code across 10 files and 28 different code blocks.

Tutorial

The study session began with the experimenter showing the participant how to use each feature in Meta-Manager. This included an explanation of the visualization (including how to zoom in to the visualization), how to use the scrubber to move through the code versions, how to search from both within the code editor and within a code box, how to filter to view only copy events, paste events, or paste events from online, and how to view each corresponding copy and paste between code versions. This tutorial was done in one of the files within the created code base, such that the participant could see and understand the context of the code base, but none of the code history task questions related to anything in that particular file.

Task

For the main task, each participant was required to use Meta-Manager to answer 8 questions. Each question was designed such that it would represent at least one aspect of a hard-to-answer question we are interested in (see Section 6.2.1) and would require the participant to use some feature of Meta-Manager to answer. Table 6.1 lists each question, along with the steps a participant could do in Meta-Manager to answer the question. The solution in the table represents the most efficient way to answer a question, but each question can be answered using other methods. For example, for question Q6, a participant can choose to search forwards in time on the AI generated code, which will pull out instances where that code changed, but they can also scrub through the code versions using the scrubber up to the present to visually see how the code is edited over time. Participants had 10 minutes per question and were not allowed to edit or run the code, or search for information online. When a participant felt they had come to an answer, they were instructed to state their answer and they would move on to the next question.

The study was designed to increase in difficulty as participants moved into files with more and more versions to filter and search through. Questions 1 and 2 were in a file with 90 versions, Question 3 and 4 were in a file with 619 versions, Question 5 was in a file with 727 versions, and questions 6 through 8 were in a file with 1,302 versions.

Analysis

For each participant, we recorded whether or not they got the correct answer for each question and how long it took them to come to the answer. “Correctness” was determined objectively by whether or not they found the correct code or code version that contained the answer and whether the participant’s summation of what they learned was accurate. If the participant did not finish within 10 minutes, the question was marked as incorrect. We additionally reviewed the video recordings to see what features of the tool and strategies participants used when coming to an answer.

Question in Task	Question from Literature	Solution
Q1. In config.ts, there is a regex for search pattern matching. Can you tell me what it is matching on and why?	Why is this code written the way it is? [83, 91, 146]	Find paste from ChatGPT, read user's ChatGPT query
Q2. There is a bug in the commented out `Promise` code. Can you find where the bug was and what happened?	How has this code changed over time? [91, 146]	Find where `Promise` was initially commented out and look at versions before that event
Q3. Prior to using `parseHTML`, the author was using a different API - what was it and why did they stop using it?	Why is this code written the way it is? [83, 91, 146]	Search backwards through time to when `parseHTML` no longer exists, see what code was there before, and see Stack Overflow post about the problem with the prior API method
Q4. Recently, some code was added to `search` that came from a different file – can you find that code and explain what changed?	Where did this code come from? [146]	Filter to see pasted code, find paste event with code copied from a different file, then search for that code in the file
Q5. Look at lines 68 to 70 – there is a commented out `forEach` loop. Can you find the last time it was used and explain why it was removed and what it was replaced with?	What code is related to this code? [36, 91, 146]	Search on commented out code, click on "Commented Out" event, find Stack Overflow post near event with replacement code
Q6. What code was generated by an AI system and what ended up happening to it?	Where did this code come from? [146]	Filter to see ChatGPT code, then search forwards in time on that code
Q7. What were all the different things that the programmer tried when setting the `match` variable?	How has this code changed over time? [91, 146]	Search backwards in time on `match`, look at events
Q8. Some code from `activate` was moved into a different file. When did this happen and what was the code that was moved?	Where did this code come from? [146]	Filter to code copied in `activate`, then see corresponding paste locations

TABLE 6.1: Each question that was asked during the task, along with what "hard-to-answer" question from prior literature it corresponds to, and the steps taken in Meta-Manager to answer the question. Note that some questions represent more than one hard-to-answer question, such as Q5, which both asks what code is related to the commented out loop, but also why the loop was commented out, which is in line with "why is this code written the way it is".

6.3.2 Participants

We recruited 7 participants (6 men, 1 woman) using study recruitment channels at our institution, along with advertisements on our social networks. All of the participants were required to have some amount of experience using TypeScript and be familliar with Visual Studio Code. Participant occupations included 4 professional software engineers, 2 researchers, and a financial operations engineer with a computer science background. The average amount of years of professional software engineering was 3.16, self-reported competency with JavaScript was 4.5 (out of 7, where 7 is expert), and an average self-reported competency score of 3 for TypeScript. All study sessions were completed and recorded using Zoom and participants used Zoom to take remote control of the experimenter's computer in order to use the tool. Participants 1 through 7 are hereafter referred to as P1 through P7.

6.3.3 Results and Discussion

Table 6.2 breaks down how often each question was solved and how long, on average, getting the correct answer to the question took. Questions 3, 4, 5, and 8 were answered correctly by all participants and did not take relatively long to solve. Participants also solved these questions in the most consistent manner, with all participants starting with the same first step that was outlined in Table 6.1 as the intended solution path. Notably, these questions correspond to 3 of the 4 types of hard-to-answer questions discussed in Section 6.2.1, suggesting that the tool was successful in supporting questions including "why is this code written the way it is", "where did this code come from", and "what code is related to this code". Additionally, in our post-task survey, participants rated Q5 as the most similar to frequently asked questions they have, suggesting that our tool's ability to answer that question is particularly valuable.

Question	Outcome	Time Spent
Q1. In config.ts, there is a regex for search pattern matching. Can you tell me what it is matching on and why?	6 correct, 1 incorrect	3 minutes 28 seconds *(min: 58 seconds, max: 6 minutes, 32 seconds, std. dev.: 1 minutes, 59 seconds)*
Q2. There is a bug in the commented out `Promise` code. Can you find where the bug was and what happened?	4 correct, 3 out-of-time	5 minutes 4 seconds *(min: 1 minute, 58 seconds, max: 7 minutes, 2 seconds, std. dev.: 2 minutes, 24 seconds)*
Q3. Prior to using `parseHTML`, the author was using a different API - what was it and why did they stop using it?	7 correct	4 minutes 35 seconds *(min: 2 minutes 23 seconds, max: 7 minutes, 38 seconds, std. dev.: 1 minute, 46 seconds)*
Q4. Recently, some code was added that came from a different file – can you find that code and explain what changed?	7 correct	4 minutes 47 seconds *(min: 2 minutes, max: 8 minutes, 8 seconds, std. dev.: 2 minutes, 29 seconds)*
Q5. Look at lines 68 to 70 – there is a commented out `forEach` loop. Can you find the last time it was used and explain why it was removed and what it was replaced with?	7 correct	4 minutes 24 seconds *(min: 1 minutes 57 seconds, max: 7 minutes, 12 seconds, std. dev.: 1 minute, 44 seconds)*
Q6. What code was generated by an AI system and what ended up happening to it?	5 correct, 2 out-of-time	6 minutes 36 seconds *(min: 2 minutes 2 seconds, max: 9 minutes, 15 seconds, std. dev.: 1 minutes, 53 seconds)*
Q7. What were all the different things that the programmer tried when setting the `match` variable?	5 correct, 2 incorrect	5 minutes 18 seconds *(min: 2 minutes 15 seconds, max: 8 minutes, 17 seconds, std. dev.: 2 minutes, 15 seconds)*
Q8. Some code from `activate` was moved into a different file. When did this happen and what was the code that was moved?	7 correct	3 minutes 42 seconds *(min: 1 minutes 14 seconds, max: 8 minutes, 35 seconds, std. dev.: 2 minutes, 17 seconds)*

TABLE 6.2: Each question, its outcome and the average amount of time spent on it.

Participants varied in how long they took to answer each question, even in cases where the majority of participants eventually did come to the correct solution. This usually occurred when participants took unintended solution paths or had some confusion about how the tool or parts of it worked, suggesting that the design of the tool could be improved to make the intended functionality more clear. For example, with Q8, P5 chose to filter for code that had been pasted into another file (i.e., clicked "See Pasted Code"), but, since the question was asking for instances in which code was *copied* (i.e., participants should click "See Copied Code"), this search did not work. P5 also noted in their post-task survey that they typically consider copy-paste as a single event, so, given how the tool treats them as two separate but connected events, they experienced some confusion. In contrast, participants following the most "optimal" path typically performed the task within 1 to 3 minutes. Given that all of these questions were designed to be complex enough that they could not be solved with one button click or search query in order to make them more realistic and, since participants lacked any sort of contextual knowledge about the code base to help guide them towards solutions, these results suggest that the tool can be a practical way of answering these types of questions.

Notably, some questions were more challenging than others. Q2 and Q6 had the lowest amount of correct responses and Q6, on average, took the longest amount of time to solve when participants did come to the correct solution. With Q6, all participants succeeded in identifying the code that came from ChatGPT but had a harder time answering the second half of the question. Participants almost always chose to search on the ChatGPT code across time but, given that the code was a large function that had many edits applied to it, it was difficult to isolate what specifically happened to the code, given their lack of context about the code base and how its various functions are used. This issue of a larger lack of context was also why Q2 was challenging – since the participants did not know how the code was used in the larger context of the project, it was difficult to isolate what part of the code contained the bug. This suggests that Meta-Manager may be improved by supporting more views that show how code is related to other parts of the code base outside of the user's current file.

In terms of strategies, participants most often utilized the search feature as their primary way of navigating through the timeline. This is not particularly surprising, given the ubiquity of search in navigation, especially code navigation, and participants were able to use that strategy to solve their questions in most cases where it was the intended solution. Note, however, that this search is backwards through time and not directly through the current text like other familiar searches, but given practice using the across-time search, participants were able to use it to support their different types of questions. Specifically, Q3 and Q5 both require participants to search backwards across time to find when `parseHTML` and a `forEach` loop, respectively, were no longer used, and all participants were able to use the search to successfully answer those questions.

Participants most often started their search using the context menu item in the code editor windows, but would later switch to using the search from within a code version in the Meta-Manager plugin user interface after finding a particular part of the code they wanted to further drill down on. For example, Q3 is solvable using this strategy by initially finding the instance in which `parseHTML` disappears and the older API method, `JSDOM`, appears. 3 of the 7 participants used this method for answering Q3. However, this question was also solvable by finding the point in time at which the API change is made, then filtering in that time span for paste events or Stack Overflow searches, as both of those events happened around the same time as the switch – 4 participants were able to solve this question using that approach. In general, the timeline labels were commonly used as a means of navigating through the code history, with most participants opting to look only at events that were marked along the timeline, and not bothering to look through versions in between these labeled points.

In the post-task survey, participants reacted favorably to Meta-Manager. Participants agreed that they would find Meta-Manager useful for their daily work (avg. 6.14 out of 7, with 7 being "strongly agree") and enjoyed the features provided by Meta-Manager (avg. 6.57 out of 7). Participants particularly liked the ability to see where code from online came from within the context of the IDE as a way to see what the original developer was doing, with one participant stating that they imagined that this will be how they spend "most of their development time in the future, with more code coming from AI" (P7). Participants desired improvements to the user interface to make some of their interactions a bit more clear (avg., 4.3 out of 7), especially with respect to the interaction between the filter, search, and zoom operations. Participants sometimes lost track of what filters they had in place when searching, and vice versa, which caused confusion. Adopting the UI for sorting and filtering from popular shopping websites might help alleviate these problems.

We additionally asked participants to rate each question asked in the study by how often they have encountered similar questions in their own programming experiences on a 5-point scale from "never ask" to "always ask" (Figure 6.4). Participants reported asking questions similar to Q5, which asked about why some code was introduced to replace some other code, most often, with 4 participants stating they "always ask" questions like this. Notably, that is also one of the questions all participants were successfully able to answer, which suggests the tool is useful in answering this type of common, hard-to-answer question. Only two questions had some participants state they never asked that question, which were the questions corresponding to reasoning about where some code originated from (an AI system, in this case) and what the previous developer had tried when implementing some change. All questions had at least one participant say that they sometimes ask that

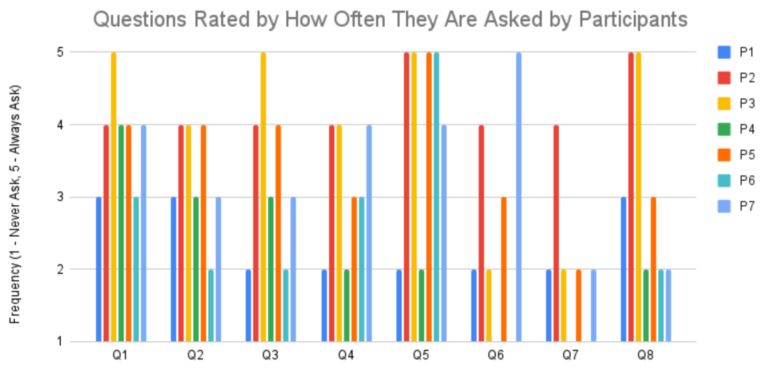


FIGURE 6.4: Each question scored by participants in terms of how often they encounter similar questions in their own programming experiences.

question, which is both inline with prior research and adds more evidence that supporting answering these questions is useful.

6.4 Future Work

Our lab study provided some evidence that Meta-Manager helps developers answer hard-to-answer questions about code. However, this was in the context of a developer joining an unfamiliar code base with no real contextual knowledge of the code or its history — while this allowed us to best assess how well the system works in assisting developers in answering these questions in, arguably, the most difficult situation, future work would benefit from seeing how Meta-Manager helps developers when they are working on their *own* code. Open questions remain in this situation — given developers' own mental models of their code base and, most likely, its history, one can imagine that usage of Meta-Manager may change, as developers' questions about the code base may become more specific since they have more information to work off of. Improvements to Meta-Manager to support more personalized information may be a richer querying system that supports project-specific terms or time constraints, such as "show me all edits to this function between bug fix #124 and now", or allowing users to define their own "events" outside of copy-paste, code commenting, etc. that the system will automatically log as an event of interest. Prior work has supported similar team and project-specific tagging of information in software projects to help with source code navigation [154, 161] – extracting project-specific tagged events as timeline events may also help developers with navigating between code versions.

Additionally, there is existing meta-information about code project history that Meta-Manager does not currently show, such as the boundaries between Git commits along the timeline, or event labels for typical historical events like merged-in pull requests. One can imagine changing the visualization to show visual boundaries between these events, which may also help to better visually convey to a user why some block of code has appeared or disappeared as code commits are merged together, pulled in, or checked out. This may also help make this information more easily navigable, considering foraging through many Git events is a known challenge [75]. We additionally want to support more event capturing, such as when documentation-type code comments are edited,[6] and other interesting code sources, such as CoPilot. CoPilot, in particular, may be particularly interesting for logging events with given that user's edits that prompt CoPilot (especially in the case of natural language prompts to CoPilot to generate some code) may serve as strong signals for what the code author was trying to do. Conveying more of this meta-information in conjunction with more structured time information may give developers more useful navigational way points for searching for code across time. Expanding the types of meta-information Meta-Manager supports may help with answering other types of programming-related questions, which the proposed work will explore.

Currently, Meta-Manager does not support saving or sharing specific code versions, queries, or filter settings. There may be situations in which it would be useful to keep track of that information, such as for communicating with collaborators about how and when a bug was introduced [92] or for saving a code version that a developer may be considering reverting back to [75, 76]. This meta-meta information (meta information about the use of the meta information about the code) could be useful to help others perform similar sensemaking to the current user, based on research [47] that multiple people through time often need to repeat previous people's work.

[6]In the current version of Meta-Manager, one can search on a code comment backwards through time in the same way that regular code can be searched on, but better categorizing the type of code comments, such as "TODO's" or "bugs", may make this information more comprehensible and searchable.

Chapter 7

Proposed Work

7.1 Introduction

In this work, we have investigated methods for improving comprehensibility of code and code-related meta-information through strengthening the relationship between these two rich information sources. We have explored the benefits of bridging code and its meta-information across different sensemaking domains including understanding API documentation and comprehending unfamiliar code, and the results have been promising with developers successfully creating and using the contextualized information to perform better on development tasks and answer relevant questions about code.

Despite this success, I recognize that we are currently in a transitory state of software engineering with the rise of AI-generated code which naturally leads to a new and growing class of meta-information. Recent reports claim that around 31% or more new code is generated from AI tools [95] and it is expected that this number will continue to grow as AI tools continually improve and the barriers to using them are lowered. With the rapidly growing capability of these tools, there is reason to believe that more developer effort will go towards prompting these tools and reviewing the generated code – a context in which additional meta-information about the generation of this code may be helpful.

The systems developed during my thesis all captured different types of meta-information in different development contexts. This includes understanding and using API documentation (Chapter 3), making sense of code (Chapters 4 and 6), and authoring and maintaining project documentation (Chapter 5). These systems worked well in isolation, but there remains an open question of whether or not these different types of meta-information may complement one another for supporting new developer sensemaking activities. For example, Meta-Manager's rich history and versioning support, coupled with Sodalite's documentation maintenance algorithm may help maintainers know *when* the documentation went out-of-date – a question which would be unanswerable with either tool in isolation.

To conclude this thesis, I propose extending Meta-Manager to unify all of the types of meta-information my other tools support. This means supporting Adamite and Catseye annotations and Sodalite documentation. During this extension process, I will also explore how these new and unified forms of meta-information may assist in the comprehension of AI-generated code. In Chapter 6, Meta-Manager showed success in capturing some of AI-tool meta-information, including ChatGPT thread information to supplement ChatGPT-written code with related rationale and provenance data. However, the information provided in Meta-Manager only provides a slice of the whole iterative, prompt-driven development and debugging that developers are doing with systems like ChatGPT. As described in [52], coordinating with these systems is an iterative process in which user-intent and the AI's response are refined while the user gains a deeper understanding of both their personal problem space and how to guide the AI towards a correct solution. I hypothesize that these iterative steps can be captured and annotated in such a way that subsequent users can follow along with this iterative sensemaking between developer and AI-system to better understand how and why some code is written the way it is.

7.2 Proposed System Design

Meta-Manager has shown that code meta-information can be organized at scale by utilizing an event-driven lens, where specific events are extracted and used for assisting in navigation. I choose to extend Meta-Manager, as opposed to one of the other systems, as a base system because it is the only system in place which directly addresses the challenge of scale, meaning that adding new types of information to it should not drastically lower the usability of the tool, which may not be the case for the other systems. Further, given that Meta-Manager treats information as events over time, many of the features in my other tools can be treated as "events", such as creating a new Sodalite code story. I plan on extending this design to support new events that are relevant to the newly-integrated systems and workflows.

7.2.1 Visualizing Meta-Information

In its current form, the Meta-Manager visualizes and presents its information as a stream along a linear timeline, with colors in the stream corresponding to code blocks. This visualization approach made sense for the Meta-Manager as its primary purpose was to collect and present meta-information over time. However, with the addition of new types of meta-information, it is unclear whether this singular visualization is still the most appropriate choice. While annotation creation, for example, can be shown as an event along the timeline, if a developer is hoping to compose multiple annotations across multiple code patches, the time of these various versions and events may become less relevant in favor of the relationship between the annotations. In such a case, it may be more appropriate to present the annotations as, e.g., a group or list of annotations, divorced from the timeline view. Other views that may make sense are a branch view as a way of clustering and showing the progression of how some code and its meta-information changed given significant edits and a "workspace" view where this curation can occur. I propose exploring the design space of how to display different forms of meta-information given the type of meta-information and developer task.

7.2.2 Supporting Catseye Code Annotations

Catseye uses annotations to assist developers in tracking code-related information. Choosing to annotate some code is an obvious signal that Meta-Manager can leverage and display on the timeline. Given an annotation's content, an annotation may be able to answer some of the questions the original Meta-Manager helped answer, such as why some code is the way it is. Catseye and Meta-Manager had largely similar design goals but took different approaches, with Catseye focusing on developer-authored and tracked insights about code during active development while Meta-Manager utilized code history to answer provenance-related questions. Both approaches relied upon associating some meta-information with the source code, thus, Catseye and Meta-Manager meta-information can be merged given whether or not they share the same source code at some point in time.

Meta-Manager can further support Catseye annotations through lowering some of the information collection cost. Meta-Manager automatically tracks blocks of code and their change history through leveraging the TypeScript AST – Catseye code annotations could utilize this same information tracking "change buffer" for capturing not only meta-information that Meta-Manager already supports (e.g., copy-pastes) but also for information that Catseye previously required users to manually track. This includes intermittent code versions, which are already automatically tracked with Meta-Manager, and the versions' associated outputs' (which can be gathered using the Visual Studio Code Debugging API). These annotation histories could then be treated in Meta-Manager the same way code blocks are, including being visualized on the timeline, tracking their own copy-paste events, etc.

A key aspect of Catseye annotations is supporting code navigation. As previously discussed in Chapter 6, Meta-Manager could be improved through supporting cross-file history exploration to understand and relate different pieces of code – Catseye annotations, especially those which have multiple anchors across files, may be another strong signal of related code that should be explored together in a shared visualization.

7.2.3 Supporting Sodalite Documentation

Sodalite allows for in-editor documentation authoring, utilizing code links for contextualizing the documentation, and maintenance, in which the links are used to assess the validity of the text. Like Catseye and Meta-Manager, the developer's source code is utilized for grounding this meta-information, thus the source code can act as the point of unification for all of these tools.

Sodalite's code links, like Catseye's code annotations, can be tracked in Meta-Manager's history system. Through utilizing Meta-Manager's saved history, we can provide maintainers and readers of documentation even more meta-information than what is currently supported. For instance, we can show how the documented code changed over time, including how similar it is to when it was initially documented and/or last updated, when it became no-longer valid, if Sodalite found the link to be invalid, who introduced the changes that "broke" the link, and so on. In this way, Sodalite and Meta-Manager complement each other well – Meta-Manager's historical information "fills in the gaps" that Sodalite previously inferred when determining out-of-datedness, while Sodalite provides the additional human-generated information about the code that Meta-Manager previously could only obtain when there happened to be web-based searches.

7.2.4 Supporting Adamite Annotations

Adamite is the only system that currently does *not* leverage the developer's source code, thus making its integration less straightforward than the previous two systems. Given that Adamite and Meta-Manager share the same FireStore instance (thus, the same authentication system), one idea is to treat Adamite annotations as events within Meta-Manager and associate the annotations with whatever code was edited before and after the user made their Adamite annotation, where the authenticated user serves as the connective property. This approach is obviously not foolproof – for example, if the user annotates something unrelated to programming, such as a review of a new camera, it would not make sense to associate that annotation with any code, even if the user did write some code around the same time that they made the annotation. To avoid this problem, I can envision improving

Adamite's text anchoring capabilities to take into account the type of web content that is annotated. Text that is wrapped in `<pre />` tags or `<code />` tags is likely code, which may have related code in the user's IDE, especially in the case of API documentation, in which it is possible to use the parsed AST to find instances in which the annotated API method(s) are used. The Adamite annotation content could then be associated with the relevant code, along with any user-added tags or annotation types. The Adamite information could also complement any web-based copy-pasted code, which Meta-Manager already supports. This bi-directional relationship between the code editor and web documentation can be further explored through extending Adamite to create annotations on code copied from the web that point *back* to where that copied code is in the developer's own project, which can be used as examples of how the API is used in "real" a project.

7.2.5 Supporting AI-Generated Code

Some new forms of meta-information are the prompts used to generate some code (ChatGPT, CoPilot), the AI-generated response (ChatGPT), and alternative code snippets that the developer did not choose (CoPilot). Meta-Manager currently captures ChatGPT prompts and responses, along with the thread URL, but does not capture or associate the edits the developer makes to that code with any future prompts that are made to refine the subsequent code to what the developer actually needs. Oftentimes, using ChatGPT or other chat-based AI systems requires the user to refine their prompts, try out whatever solution the AI provides, and repeat until a satisfactory solution is achieved. During this process, the developer both refines their mental model of the problem space they are in, and, as a byproduct of this refinement process, creates a log of what they tried and what did and did not work while implementing their solution. We can improve the Meta-Manager to better illustrate this iterative process as an explorable "sub-history" to further help with answering questions like "why is this code written the way it is", while also answering a new class of "hard-to-answer" questions such as "how thoroughly was this code tested" [146], which is a common concern with AI-generated code [97]. Meta-Manager also does not currently annotate code generated by CoPilot as AI-generated code along its history timeline, which we plan to rectify, along with capturing the prompt that instigated CoPilot and the alternative solutions that were not chosen, which may serve as useful meta-information for better reasoning about the correctness of the code and what the developer was attempting to do when the code was written.

7.3 Evaluation Plan

I plan to conduct a controlled lab study to assess the utility, usability, and effectiveness of this new, integrated system in helping developers make sense of unfamiliar code. I would like to compare the system to both the original Meta-Manager and to a repository with no additional tooling or information beyond what is present in a "typical" repository (e.g., a README file, code comments, Git history, etc.). Additionally, I am interested in how this rich history can help developers when working with their *own* code. To that end, I would like to deploy the new system as part of a supplementary field study. The controlled lab study can provide evidence in terms of how correctly developers can answer the history and rationale questions we are interested in, while the field study can complement that information with how well the system performs in helping developers answer their own questions as the system builds up a larger and richer history of their code base. I am particularly interested in seeing how developers' nascent mental models of unfamiliar code versus more robust mental models change how they interact with the tool and whether these different use cases produce different questions. With these studies, I plan to investigate the following research questions:

- Can the unified forms of meta-information about code answer developers' questions about *unfamiliar* code?
- Can the unified forms of meta-information about code answer developers' questions about their *own* code?
- How well does each type of supported meta-information map to known hard-to-answer questions about code?
- How well do the integrated systems work together in supporting question-answering in comparison to traditional methods?

7.4 Timeline of Completion

My goal is to complete my dissertation by August 2024. My proposed schedule is shown below:

- December 6th, 2023: Thesis Proposal
- December 2023 - March 2024: Design and build proposed system
- March 2024 - May 2024: Run lab studies for the proposed system
- January 2024 - June 2024: Job search
- June 2024 - August 2024: Thesis writing and defense

Bibliography

[1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. "Supporting code comprehension via annotations: Right information at the right time and place". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2020, pp. 1–10.

[2] Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, and Michele Lanza. "Automated documentation of android apps". In: *IEEE Transactions on Software Engineering* 47.1 (2019), pp. 204–220.

[3] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. "Software Documentation: The Practitioners' Perspective". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, 590–601. ISBN: 9781450371216. DOI: 10.1145/3377811.3380405. URL: https://doi.org/10.1145/3377811.3380405.

[4] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[5] Maristella Agosti, Giorgetta Bonfiglio-Dosio, and Nicola Ferro. "A historical and contemporary study on annotations to derive key features for systems design". In: *International Journal on Digital Libraries* 8.1 (2007), pp. 1–19.

[6] Mohammad Allahbakhsh, Boualem Benatallah, Aleksandar Ignjatovic, Hamid Reza Motahari-Nezhad, Elisa Bertino, and Schahram Dustdar. "Quality Control in Crowdsourcing Systems: Issues and Directions". In: *IEEE Internet Computing* 17 (2 2013), pp. 76–81. DOI: 10.1109/MIC.2013.20. URL: https://doi.org/10.1109/MIC.2013.20.

[7] Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K. Roy, and Kevin A. Schneider. "Answering Questions about Unanswered Questions of Stack Overflow". In: *MSR 2013*. San Francisco, CA, USA: IEEE, 2013, pp. 97–100. DOI: 10.1109/MSR.2013.6624015. URL: https://doi.org/10.1109/MSR.2013.6624015.

[8] Alberto Bacchelli and Christian Bird. "Expectations, outcomes, and challenges of modern code review". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[9] Sebastian Baltes, Richard Kiefer, and Stephan Diehl. "Attribution Required: Stack Overflow Code Snippets in GitHub Projects". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 161–163. DOI: 10.1109/ICSE-C.2017.99.

[10] Sebastian Baltes, Christoph Treude, and Stephan Diehl. "SOTorrent: Studying the Origin, Evolution, and Usage of Stack Overflow Code Snippets". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 191–194. DOI: 10.1109/MSR.2019.00038.

[11] Lingfeng Bao, Deheng Ye, Zhenchang Xing, Xin Xia, and Xinyu Wang. "Activityspace: a remembrance framework to support interapplication information needs". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 864–869.

[12] David Bawden, Clive Holtham, and Nigel Courtney. "Perspectives on information overload". In: *Aslib proceedings*. Vol. 51. 8. MCB UP Ltd. 1999, pp. 249–255.

[13] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. "Codebook: Discovering and Exploiting Relationships in Software Repositories". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, 125–134. ISBN: 9781605587196. DOI: 10.1145/1806799.1806821. URL: https://doi.org/10.1145/1806799.1806821.

[14] Andrew Begel and Beth Simon. "Novice Software Developers, All over Again". In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. Sydney, Australia: Association for Computing Machinery, 2008, 3–14. ISBN: 9781605582160. DOI: 10.1145/1404520.1404522. URL: https://doi.org/10.1145/1404520.1404522.

[15] Michael Bernstein, Max Van Kleek, David Karger, and MC Schraefel. "Information scraps: How and why information eludes our personal information management tools". In: *ACM Transactions on Information Systems (TOIS)* 26.4 (2008), pp. 1–46.

[16] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. "What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 308–318.

[17] Jürgen Börstler and Barbara Paech. "The role of method chains and comments in software readability and comprehension—An experiment". In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 886–898.

[18] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. "Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 2503–2512. ISBN: 9781605589299. DOI: 10.1145/1753326.1753706. URL: https://doi.org/10.1145/1753326.1753706.

[19] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. "Example-Centric Programming: Integrating Web Search into the Development Environment". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 513–522. ISBN: 9781605589299. DOI: 10.1145/1753326.1753402. URL: https://doi.org/10.1145/1753326.1753402.

[20] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code”. In: *CHI ’09*. CHI ’09. Boston, MA, USA: Association for Computing Machinery, 2009, 1589–1598. ISBN: 9781605582467. DOI: 10.1145/1518701.1518944. URL: https://doi.org/10.1145/1518701.1518944.

[21] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. “GenderMag: A Method for Evaluating Software’s Gender Inclusiveness”. In: *Interacting with Computers* 28.6 (Nov. 2016). DOI: 10.1145/3134737. URL: https://doi.org/10.1145/3134737.

[22] CodeSandbox BV. *CodeSandbox: Online Code Editor and IDE for Rapid Web Development*. CodeSandbox BV. 2021. URL: https://codesandbox.io/.

[23] Paul Chandler and John Sweller. “Cognitive load theory and the format of instruction”. In: *Cognition and instruction* 8.4 (1991), pp. 293–332.

[24] Joseph Chee Chang, Nathan Hahn, Yongsung Kim, Julina Coupland, Bradley Breneisen, Hannah S Kim, John Hwong, and Aniket Kittur. “When the Tab Comes Due:Challenges in the Cost Structure of Browser Tab Usage”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445585. URL: https://doi.org/10.1145/3411764.3445585.

[25] Joseph Chee Chang, Nathan Hahn, and Aniket Kittur. “Supporting Mobile Sensemaking Through Intentionally Uncertain Highlighting”. In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST ’16. Tokyo, Japan: Association for Computing Machinery, 2016, 61–68. ISBN: 9781450341899. DOI: 10.1145/2984511.2984538. URL: https://doi.org/10.1145/2984511.2984538.

[26] Preetha Chatterjee, Manziba Akanda Nishi, Kostadin Damevski, Vinay Augustine, Lori Pollock, and Nicholas A. Kraft. “What information about code snippets is available in different software-related documents? An exploratory study”. In: *SANER 2017*. New York City, NY, USA: IEEE, 2017, pp. 382–386.

[27] J. C. Chen and S. J. Huang. “An empirical analysis of the impact of software development problem factors on software maintainability”. In: *Journal of Systems and Software* 82.6 (2009).

[28] Kai Chen, Stephen R. Schach, Liguo Yu, Jeff Offutt, and Gillian Z. Heller. “Open-Source Change Logs”. In: *Empirical Software Engineering* 9.3 (Sept. 2004), pp. 197–210. ISSN: 1573-7616. DOI: 10.1023/B:EMSE.0000027779.70556.d0. URL: https://doi.org/10.1023/B:EMSE.0000027779.70556.d0.

[29] 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*. CHI ’07. San Jose, California, USA: Association for Computing Machinery, 2007, 557–566. ISBN: 9781595935939. DOI: 10.1145/1240624.1240714. URL: https://doi.org/10.1145/1240624.1240714.

[30] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. “Self-explanations: How students study and use examples in learning to solve problems”. In: *Cognitive science* 13.2 (1989), pp. 145–182.

[31] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. “Self-explanations: How students study and use examples in learning to solve problems”. In: *Cognitive science* 13.2 (1989), pp. 145–182.

[32] Michael J. Coblenz, Amy J. Ko, and Brad A. Myers. “JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks”. In: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*. eclipse ’06. Portland, Oregon, USA: Association for Computing Machinery, 2006, 65–69. ISBN: 1595936211. DOI: 10.1145/1188835.1188849. URL: https://doi.org/10.1145/1188835.1188849.

[33] C.R. Cook, J.C. Scholtz, and J.C. Spohrer. *Empirical Studies of Programmers: Fifth Workshop : Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA*. Human/computer interaction. Norwood, NJ, USA: Ablex Publishing Corporation, 1993. ISBN: 9781567500899. URL: https://books.google.com/books?id=rMmxq8q0CGYC.

[34] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. “Context-based search to overcome learning barriers in software development”. In: *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. 2012, pp. 47–51. DOI: 10.1109/RAISE.2012.6227970.

[35] D. Cordes and M. Brown. “The literate-programming paradigm”. In: *Computer* 24.6 (1991), pp. 52–61. DOI: 10.1109/2.86838.

[36] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. “Hipikat: a project memory for software development”. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465. DOI: 10.1109/TSE.2005.71.

[37] Alex Cummaudo, Rajesh Vasa, John Grundy, and Mohamed Abdelrazek. “Requirements of API Documentation: A Case Study Into Computer Vision Services”. In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/TSE.2020.3047088. URL: https://doi.org/10.1109/TSE.2020.3047088.

[38] Barthélémy Dagenais and Martin P. Robillard. “Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors”. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, 127–136. ISBN: 9781605587912. DOI: 10.1145/1882291.1882312. URL: https://doi.org/10.1145/1882291.1882312.

[39] Uri Dekel and James D. Herbsleb. “Reading the documentation of invoked API functions in program comprehension”. In: *2009 IEEE 17th International Conference on Program Comprehension*. New York City, NY, USA: IEEE, 2009, pp. 168–177. DOI: 10.1109/ICPC.2009.5090040. URL: https://doi.org/10.1109/10.1109/ICPC.2009.5090040.

[40] Robert Deline, Mary Czerwinski, and George Robertson. “Easing program comprehension by sharing navigation data”. In: *VLHCC 2005*. New York City, NY, USA: IEEE, 2005, pp. 241–248. DOI: 10.1109/VLHCC.2005.32. URL: https://doi.org/10.1109/VLHCC.2005.32.

[41] Google Developers. *Cloud Firestore: Store and sync app data at global scale*. Google LLC. 2022. URL: https://firebase.google.com/products/firestore.

[42] Ekwa Duala-Ekoko and Martin P. Robillard. "Asking and answering questions about unfamiliar APIs: An exploratory study". In: *ICSE 2012*. New York City, NY, USA: IEEE, 2012, pp. 266–276.

[43] Daniel S. Eisenberg, Jeffrey Stylos, and Brad A. Myers. "Apatite: A New Interface for Exploring APIs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 1331–1334. ISBN: 9781605589299. DOI: 10.1145/1753326.1753525. URL: https://doi.org/10.1145/1753326.1753525.

[44] Elastic. *Free and Open Search: Elasticsearch*. Elastic. 2021. URL: https://www.elastic.co/.

[45] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. "How do API documentation and static typing affect API usability?" In: *ICSE 2014*. New York City, NY, USA: ACM, 2014, pp. 632–642. DOI: 10.1145/2568225.2568299. URL: https://doi.org/10.1145/2568225.2568299.

[46] Facebook. *React - A JavaScript library for building user interfaces*. 2023. URL: https://reactjs.org/.

[47] Kristie Fisher, Scott Counts, and Aniket Kittur. "Distributed Sensemaking: Improving Sensemaking by Leveraging the Efforts of Previous Users". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 247–256. ISBN: 9781450310154. DOI: 10.1145/2207676.2207711. URL: https://doi.org/10.1145/2207676.2207711.

[48] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks". In: *ACM Trans. Softw. Eng. Methodol.* 22.2 (2013). ISSN: 1049-331X. DOI: 10.1145/2430545.2430551. URL: https://doi.org/10.1145/2430545.2430551.

[49] Beat Fluri, Michael Wursch, and Harald C Gall. "Do code and comments co-evolve? on the relation between source code and comment changes". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE. 2007, pp. 70–79.

[50] Andrew Forward and Timothy C. Lethbridge. "The Relevance of Software Documentation, Tools and Technologies: A Survey". In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. DocEng '02. McLean, Virginia, USA: Association for Computing Machinery, 2002, 26–33. ISBN: 1581135947. DOI: 10.1145/585058.585065. URL: https://doi.org/10.1145/585058.585065.

[51] Thomas Fritz and Gail C Murphy. "Using information fragments to answer the questions developers ask". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 175–184.

[52] Elena L. Glassman. *Designing Interfaces for Human-Computer Communication: An On-Going Collection of Considerations*. 2023. arXiv: 2309.02257 [cs.HC].

[53] Sanuri Dananja Gunawardena, Peter Devine, Isabelle Beaumont, Lola Piper Garden, Emerson Murphy-Hill, and Kelly Blincoe. "Destructive Criticism in Software Code Review Impacts Inclusion". In: *Proc. ACM Hum.-Comput. Interact.* 6.CSCW2 (2022). DOI: 10.1145/3555183. URL: https://doi.org/10.1145/3555183.

[54] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. "Collective Code Bookmarks for Program Comprehension". In: *2011 IEEE 19th International Conference on Program Comprehension*. 2011, pp. 101–110. DOI: 10.1109/ICPC.2011.19.

[55] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. "On the use of automated text summarization techniques for summarizing source code". In: *2010 17th Working Conference on Reverse Engineering*. IEEE. 2010, pp. 35–44.

[56] Björn Hartmann, Mark Dhillon, and Matthew K. Chan. "HyperSource: Bridging the Gap between Source and Code-Related Web Sites". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 2207–2210. ISBN: 9781450302289. DOI: 10.1145/1978942.1979263. URL: https://doi.org/10.1145/1978942.1979263.

[57] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. "9.6 Million Links in Source Code Comments: Purpose, Evolution, and Decay". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1211–1221. DOI: 10.1109/ICSE.2019.00123.

[58] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. "Interactive Extraction of Examples from Existing Code". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3173659. URL: https://doi.org/10.1145/3173574.3173659.

[59] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. "Managing Messes in Computational Notebooks". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–12. ISBN: 9781450359702. DOI: 10.1145/3290605.3300500. URL: https://doi.org/10.1145/3290605.3300500.

[60] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. "Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376798. URL: https://doi.org/10.1145/3313831.3376798.

[61] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 643–653. ISBN: 9781450356381. DOI: 10.1145/3180155.3180176. URL: https://doi.org/10.1145/3180155.3180176.

[62] Ken Hinckley, Xiaojun Bi, Michel Pahud, and Bill Buxton. "Informal Information Gathering Techniques for Active Reading". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1893–1896. ISBN: 9781450310154. DOI: 10.1145/2207676.2208327. URL: https://doi.org/10.1145/2207676.2208327.

[63] Reid Holmes and Andrew Begel. “Deep Intellisense: A Tool for Rehydrating Evaporated Information”. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. Leipzig, Germany: Association for Computing Machinery, 2008, 23–26. ISBN: 9781605580241. DOI: 10.1145/1370750.1370755. URL: https://doi.org/10.1145/1370750.1370755.

[64] Amber Horvath, Sachin Grover, Sihan Dong, Emily Zhou, Finn Voichick, Mary Beth Kery, Shwetha Shinju, Daye Nam, Mariann Nagy, and Brad Myers. “The Long Tail: Understanding the Discoverability of API Functionality”. In: *VLHCC 2019*. New York, NY, USA: IEEE, 2019, pp. 157–161. DOI: 10.1109/VLHCC.2019.8818681. URL: https://doi.org/10.1109/VLHCC.2019.8818681.

[65] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. “Understanding How Programmers Can Use Annotations on Documentation”. In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI ’22. New Orleans, LA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3491102.3502095. URL: https://doi.org/10.1145/3491102.3502095.

[66] Amber Horvath, Andrew Macvean, and Brad A. Myers. “Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code”. In: *Insubmission*. 2024.

[67] Amber Horvath, Andrew Macvean, and Brad A. Myers. “Support for Long-Form Documentation Authoring and Maintenance”. In: *VL/HCC 2023*. 2023.

[68] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. “Using Annotations for Sensemaking About Code”. In: *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST ’22. Bend, OR, USA: Association for Computing Machinery, 2022. ISBN: 9781450393201. DOI: 10.1145/3526113.3545667. URL: https://doi.org/10.1145/3526113.3545667.

[69] Amber Horvath, Mariann Nagy, Finn Voichick, Mary Beth Kery, and Brad A Myers. “Methods for investigating mental models for learners of APIs”. In: *CHI LBW ’19*. New York, NY, USA: ACM, 2019, pp. 1–6.

[70] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. “CnP: Towards an environment for the proactive management of copy-and-paste programming”. In: *2009 IEEE 17th International Conference on Program Comprehension*. 2009, pp. 238–242. DOI: 10.1109/ICPC.2009.5090049.

[71] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. “API Method Recommendation without Worrying about the Task-API Knowledge Gap”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE ’18. Montpellier, France: Association for Computing Machinery, 2018, 293–304. ISBN: 9781450359375. DOI: 10.1145/3238147.3238191. URL: https://doi.org/10.1145/3238147.3238191.

[72] Hypothes.is. *Hypothes.is: Annotate the web, with anyone, anywhere.* Hypothes.is. 2012. URL: https://web.hypothes.is/.

[73] Peiling Jiang, Fuling Sun, and Haijun Xia. “Log-It: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI ’23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581403. URL: https://doi.org/10.1145/3544548.3581403.

[74] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. “A case study of onboarding in software teams: Tasks and strategies”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 613–623.

[75] Mary Beth Kery, Amber Horvath, and Brad Myers. “Variolite: Supporting Exploratory Programming by Data Scientists”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI ’17. Denver, Colorado, USA: Association for Computing Machinery, 2017, 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626. URL: https://doi.org/10.1145/3025453.3025626.

[76] Mary Beth Kery, Bonnie E. John, Patrick O’Flaherty, Amber Horvath, and Brad A. Myers. “Towards Effective Foraging by Data Scientists to Find Past Analysis Choices”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–13. ISBN: 9781450359702. DOI: 10.1145/3290605.3300322. URL: https://doi.org/10.1145/3290605.3300322.

[77] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. “The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–11. ISBN: 9781450356206. DOI: 10.1145/3173574.3173748. URL: https://doi.org/10.1145/3173574.3173748.

[78] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. “How Secure is Code Generated by ChatGPT?” In: *arXiv preprint arXiv:2304.09655* (2023).

[79] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. “An ethnographic study of copy and paste programming practices in OOPL”. In: *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE’04.* IEEE. 2004, pp. 83–92.

[80] Aniket Kittur, Andrew M. Peters, Abdigani Diriye, Trupti Telang, and Michael R. Bove. “Costs and benefits of structured information foraging”. In: *CHI 2013*. New York, NY, USA: ACM, 2013, pp. 2989–2998.

[81] D. E. Knuth. “Literate Programming”. In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. eprint: https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf. URL: https://doi.org/10.1093/comjnl/27.2.97.

[82] Amy J. Ko, Htet Aung, and Brad A. Myers. “Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: Association for Computing Machinery, 2005, 126–135. ISBN: 1581139632. DOI: 10.1145/1062455.1062492. URL: https://doi.org/10.1145/1062455.1062492.

[83] Amy J Ko, Robert DeLine, and Gina Venolia. "Information needs in collocated software development teams". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 344–353.

[84] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. "A practical guide to controlled experiments of software engineering tools with human participants". In: *Empirical Software Engineering* 20.1 (2015), pp. 110–141.

[85] Amy J. Ko and Brad A. Myers. "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, 301–310. ISBN: 9781605580791. DOI: 10.1145/1368088.1368130. URL: https://doi.org/10.1145/1368088.1368130.

[86] Amy J. Ko and Brad A. Myers. "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. Vienna, Austria: Association for Computing Machinery, 2004, 151–158. ISBN: 1581137028. DOI: 10.1145/985692.985712. URL: https://doi.org/10.1145/985692.985712.

[87] Amy J. Ko and Brad A. Myers. "Finding Causes of Program Output with the Java Whyline". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. Boston, MA, USA: Association for Computing Machinery, 2009, 1569–1578. ISBN: 9781605582467. DOI: 10.1145/1518701.1518942. URL: https://doi.org/10.1145/1518701.1518942.

[88] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32.12 (2006), pp. 971–987. DOI: 10.1109/TSE.2006.116.

[89] Amy J. Ko and Bob Uttl. "Individual differences in program comprehension strategies in unfamiliar programming systems". In: *11th Annual Workshop on Program Comprehension*. New York, NY, USA: IEEE, 2003, pp. 175–184. DOI: 10.1109/WPC.2003.1199201. URL: https://doi.org/10.1109/WPC.2003.1199201.

[90] 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*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, 1028–1038. ISBN: 9781450339001. DOI: 10.1145/2884781.2884840. URL: https://doi.org/10.1145/2884781.2884840.

[91] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. "Program comprehension as fact finding". In: *ESEC-FSE 2007*. New York, NY, USA: ACM, 2007, pp. 361–270.

[92] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. "How Programmers Debug, Revisited: An Information Foraging Theory Perspective". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 197–215. DOI: 10.1109/TSE.2010.111.

[93] Timothy C. Lethbirdge, Janice Singer, and Andrew Forward. "How software engineers use documentation: the state of the practice". In: *IEEE Software* 20 (6 Nov. 2003), pp. 35–39. DOI: 10.1109/MS.2003.1241364. URL: https://doi.org/10.1109/MS.2003.1241364.

[94] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. "A Qualitative Study on the Implementation Design Decisions of Developers". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, 435–447. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00047. URL: https://doi.org/10.1109/ICSE48619.2023.00047.

[95] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". In: *Proceedings of the 46th International Conference on Software Engineering*. ICSE '24. To appear. Lisbon, Portugal, 2024. URL: arXiv:2303.17125.

[96] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. "ChangeScribe: A Tool for Automatically Generating Commit Messages". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 709–712. DOI: 10.1109/ICSE.2015.229.

[97] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation". In: *arXiv preprint arXiv:2305.01210* (2023).

[98] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. "Unakite: Scaffolding Developers' Decision-Making Using the Web". In: *UIST 2019*. New York, NY, USA: ACM, 2019, pp. 67–80.

[99] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. "Crystalline: Lowering the Cost for Developers to Collect and Organize Information for Decision Making". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3491102.3501968. URL: https://doi.org/10.1145/3491102.3501968.

[100] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. "To Reuse or Not To Reuse? A Framework and System for Evaluating Summarized Knowledge". In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW1 (2021). DOI: 10.1145/3449240. URL: https://doi.org/10.1145/3449240.

[101] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. "Automatic generation of pull request descriptions". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 176–188.

[102] Walid Maalej and Hans-Jorg Happel. "From work to word: How do software developers describe their work?" In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. 2009, pp. 121–130. DOI: 10.1109/MSR.2009.5069490.

[103] Walid Maalej and Martin P. Robillard. "Patterns of Knowledge in API Reference Documentation". In: *IEEE Transactions on Software Engineering* 39.9 (2013), pp. 1264–1282. DOI: 10.1109/TSE.2013.12.

[104] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. "On the Comprehension of Program Comprehension". In: *Transactions on Software Engineering* 23 (4 2014), pp. 1–37. DOI: 10.1145/2622669. URL: https://doi.org/10.1145/2622669.

[105] Catherine C. Marshall and Sara Bly. “Saving and Using Encountered Information: Implications for Electronic Periodicals”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’05. Portland, Oregon, USA: Association for Computing Machinery, 2005, 111–120. ISBN: 1581139985. DOI: 10.1145/1054972.1054989. URL: https://doi.org/10.1145/1054972.1054989.

[106] Anneliese von Mayrhauser and A Marie Vans. “Hypothesis-driven understanding processes during corrective maintenance of large scale software”. In: *1997 Proceedings International Conference on Software Maintenance*. IEEE. 1997, pp. 12–20.

[107] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. “Recommending Source Code Examples via API Call Usages and Documentation”. In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: Association for Computing Machinery, 2010, 21–25. ISBN: 9781605589749. DOI: 10.1145/1808920.1808925. URL: https://doi.org/10.1145/1808920.1808925.

[108] Michael Meng, Stephanie M Steinhard, and Andreas Schubert. “How developers use API documentation: an observation study”. In: *Communication Design Quarterly* 7 (2 2019), pp. 40–49. DOI: 10.1145/3358931.3358937. URL: https://doi.org/10.1145/3358931.3358937.

[109] Microsoft. *GitHub*. Microsoft. 2023. URL: https://github.com.

[110] Microsoft. *Visual Studio Code*. Microsoft. 2023. URL: https://code.visualstudio.com/.

[111] Microsoft. *Webview API | Visual Studio Code Extension API*. Microsoft. 2023. URL: https://code.visualstudio.com/api/extension-guides/webview.

[112] Gail C Murphy, Mik Kersten, Martin P Robillard, and Davor Čubranić. “The emergent structure of development tasks”. In: *European Conference on Object-Oriented Programming*. Springer. 2005, pp. 33–48.

[113] Emerson Murphy-Hill, Jillian Dicker, Margaret Morrow Hodges, Carolyn D. Egelman, Ciera Jaspan, Lan Cheng, Elizabeth Kammer, Ben Holtz, Matthew A. Jorde, Andrea Knight Dolan, and Collin Green. “Engineering Impacts of Anonymous Author Code Review: A Field Experiment”. In: *IEEE Transactions on Software Engineering* 48.7 (2022), pp. 2495–2509. DOI: 10.1109/TSE.2021.3061527.

[114] Emerson Murphy-Hill, Ciera Jaspan, Carolyn Egelman, and Lan Cheng. “The Pushback Effects of Race, Ethnicity, Gender, and Age in Code Review”. In: *Commun. ACM* 65.3 (2022), 52–57. ISSN: 0001-0782. DOI: 10.1145/3474097. URL: https://doi.org/10.1145/3474097.

[115] Brad A. Myers and Jeffrey Stylos. “Improving API Usability”. In: *Communications of the ACM* 59.6 (2016), pp. 62–69. DOI: 10.1145/2896587. URL: https://doi.org/10.1145/2896587.

[116] Daye Nam, Brad Myers, Bogdan Vasilescu, and Vincent Hellendoorn. “Improving API Knowledge Discovery with ML: A Case Study of Comparable API Methods”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 1890–1906. DOI: 10.1109/ICSE48619.2023.00161.

[117] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. “What makes a good code example?: A study of programming Q&A in StackOverflow”. In: *ICSM 2012*. New York, NY, USA: IEEE, 2012, pp. 25–34.

[118] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. “What programmers really want: results of a needs assessment for SDK documentation”. In: *SIGDOC 2002*. New York, NY, USA: ACM, 2002, pp. 133–141. DOI: 10.1145/584955.584976. URL: https://doi.org/10.1145/584955.584976.

[119] Observable. *D3 by Observable | The JavaScript library for bespoke data visualization*. 2023. URL: d3js.org.

[120] Stephen Oney and Joel Brandt. “Codelets: Linking Interactive Documentation and Example Code in the Editor”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, 2697–2706. ISBN: 9781450310154. DOI: 10.1145/2207676.2208664. URL: https://doi.org/10.1145/2207676.2208664.

[121] Dennis Pagano and Walid Maalej. “How Do Developers Blog? An Exploratory Study”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, 123–132. ISBN: 9781450305747. DOI: 10.1145/1985441.1985461. URL: https://doi.org/10.1145/1985441.1985461.

[122] Dennis Pagano and Walid Maalej. “How do open source communities blog?” In: *Empirical Software Engineering* 18.6 (Dec. 2013), pp. 1090–1124. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9211-2. URL: https://doi.org/10.1007/s10664-012-9211-2.

[123] Chris Parnin and Robert DeLine. “Evaluating Cues for Resuming Interrupted Programming Tasks”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2010, 93–102. ISBN: 9781605589299. URL: https://doi.org/10.1145/1753326.1753342.

[124] Chris Parnin and Spencer Rugaber. “Resumption strategies for interrupted programming tasks”. In: *Software Quality Journal* 19.1 (2011), pp. 5–34.

[125] Chris Parnin, Christoph Treude, and Margaret-Anne Storey. “Blogging developer knowledge: Motivations, challenges, and future directions”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 211–214. DOI: 10.1109/ICPC.2013.6613850.

[126] Piling.js. *The Piling.js Docs*. Piling.js. 2021. URL: https://piling.js.org/docs/.

[127] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. “Reactive Information Foraging: An Empirical Investigation of Theory-Based Recommender Systems for Programmers”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1471–1480. ISBN: 9781450310154. DOI: 10.1145/2207676.2208608. URL: https://doi.org/10.1145/2207676.2208608.

[128] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. "To fix or to learn? How production bias affects developers' information foraging during debugging". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 11–20. DOI: 10.1109/ICSM.2015.7332447.

[129] David Piorkowski, Soya Park, April Yi Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. "How AI Developers Overcome Communication Challenges in a Multidisciplinary Team: A Case Study". In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW1 (2021). DOI: 10.1145/3449205. URL: https://doi.org/10.1145/3449205.

[130] Ben Popper and David Gibson. *How often do people actually copy and paste from Stack Overflow? Now we know.* URL: https://stackoverflow.blog/2021/12/30/how-often-do-people-actually-copy-and-paste-from-stack-overflow-now-we-know/.

[131] Aniket Potdar and Emad Shihab. "An Exploratory Study on Self-Admitted Technical Debt". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 91–100. DOI: 10.1109/ICSME.2014.31.

[132] Pooja Rani, Mathias Birrer, Sebastiano Panichella, Mohammad Ghafari, and Oscar Nierstrasz. "What do developers discuss about code comments?" In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 153–164.

[133] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. "Summarizing software artifacts: a case study of bug reports". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 505–514.

[134] Steven P. Reiss. "Tracking Source Locations". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, 11–20. ISBN: 9781605580791. DOI: 10.1145/1368088.1368091. URL: https://doi.org/10.1145/1368088.1368091.

[135] Martin P. Robillard. "Turnover-Induced Knowledge Loss in Practice". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, 1292–1302. ISBN: 9781450385626. DOI: 10.1145/3468264.3473923. URL: https://doi.org/10.1145/3468264.3473923.

[136] Martin P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE Software* 26 (6 Oct. 2009), pp. 27–34. DOI: 10.1109/MS.2009.193. URL: https://doi.org/10.1109/MS.2009.193.

[137] Martin P. Robillard and Robert DeLine. "A field study of API learning obstacles". In: *Empirical Software Engineering* 16 (6 2011), pp. 703–732.

[138] M.P. Robillard, W. Coelho, and G.C. Murphy. "How effective developers investigate source code: an exploratory study". In: *IEEE Transactions on Software Engineering* 30.12 (2004), pp. 889–903. DOI: 10.1109/TSE.2004.101.

[139] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. "How do professional developers comprehend software?" In: *ICSE 2012*. New York, NY, USA: ACM, 2012, pp. 632–542. DOI: 10.1109/ICSE.2012.6227188. URL: https://doi.org/10.1109/ICSE.2012.6227188.

[140] 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*. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 181–190. ISBN: 9781450356596. DOI: 10.1145/3183519.3183525. URL: https://doi.org/10.1145/3183519.3183525.

[141] Bill N Schilit, Gene Golovchinsky, and Morgan N Price. "Beyond paper: supporting active reading with free form digital ink annotations". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1998, pp. 249–256.

[142] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. "Analyzing Code Comments to Boost Program Comprehension". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 325–334. DOI: 10.1109/APSEC.2018.00047.

[143] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. "To document or not to document? An exploratory study on developers' motivation to document code". In: *Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27*. Springer. 2015, pp. 100–106.

[144] Ben Shneiderman. *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers, 1980.

[145] Stephen Shum and Curtis Cook. "Using Literate Programming to Teach Good Programming Practices". In: *Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education*. SIGCSE '94. Phoenix, Arizona, USA: Association for Computing Machinery, 1994, 66–70. ISBN: 0897916468. DOI: 10.1145/191029.191059. URL: https://doi.org/10.1145/191029.191059.

[146] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. "Asking and Answering Questions during a Programming Change Task". In: *IEEE Transactions on Software Engineering* 34.4 (2008), pp. 434–451. DOI: 10.1109/TSE.2008.26.

[147] Jeongju Sohn and Shin Yoo. "FLUCCS: Using Code and Change Metrics to Improve Fault Localization". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, 273–283. ISBN: 9781450350761. DOI: 10.1145/3092703.3092717. URL: https://doi.org/10.1145/3092703.3092717.

[148] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. "A Study of the Documentation Essential to Software Maintenance". In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting amp; Designing for Pervasive Information*. SIGDOC '05. Coventry, United Kingdom: Association for Computing Machinery, 2005, 68–75. ISBN: 1595931759. DOI: 10.1145/1085313.1085331. URL: https://doi.org/10.1145/1085313.1085331.

[149] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. "Foraging among an overabundance of similar variants". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 2016, pp. 3509–3521.

[150] Captain Stack. *Captain Stack - Code suggestion for VSCode*. URL: https://github.com/hieunc229/copilot-clone.

[151] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F Redmiles. “A systematic literature review on the barriers faced by newcomers to open source software projects”. In: *Information and Software Technology* 59 (2015), pp. 67–85.

[152] Christoph Johann Stettina and Werner Heijstek. “Necessary and Neglected? An Empirical Study of Internal Documentation in Agile Software Development Teams”. In: *Proceedings of the 29th ACM International Conference on Design of Communication*. SIGDOC ’11. Pisa, Italy: Association for Computing Machinery, 2011, 159–166. ISBN: 9781450309363. DOI: 10.1145/2038476.2038509. URL: https://doi.org/10.1145/2038476.2038509.

[153] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. “TODO or to bug”. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 251–260. DOI: 10.1145/1368088.1368123.

[154] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. “How Software Developers Use Tagging to Support Reminding and Refinding”. In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 470–483. DOI: 10.1109/TSE.2009.15.

[155] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. “/* iComment: Bugs or bad comments?*”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, pp. 145–158.

[156] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. “@ tcomment: Testing javadoc comments to detect comment-code inconsistencies”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 260–269.

[157] Craig S. Tashman and W. Keith Edwards. “Active Reading and Its Discontents: The Situations, Problems and Ideas of Readers”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 2927–2936. ISBN: 9781450302289. DOI: 10.1145/1978942.1979376. URL: https://doi.org/10.1145/1978942.1979376.

[158] Grace Taylor and Steven Clarke. “A Tour Through Code: Helping Developers Become Familiar with Unfamiliar Code”. In: *Psychology of Programming Interest Group 33rd Annual Workshop*. PPIG 2022, pp. 114–126.

[159] Suresh Thummalapenta and Tao Xie. “Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, 204–213. ISBN: 9781595938824. DOI: 10.1145/1321631.1321663. URL: https://doi.org/10.1145/1321631.1321663.

[160] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. “What Makes a Good Commit Message?” In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 2389–2401. ISBN: 9781450392211. DOI: 10.1145/3510003.3510205. URL: https://doi.org/10.1145/3510003.3510205.

[161] Christoph Treude and Margaret-Anne Storey. “Work Item Tagging: Communicating Concerns in Collaborative Software Development”. In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 19–34. DOI: 10.1109/TSE.2010.91.

[162] Jason Tsay, Laura Dabbish, and James Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 356–366. ISBN: 9781450327565. DOI: 10.1145/2568225.2568315. URL: https://doi.org/10.1145/2568225.2568315.

[163] Gias Uddin and Martin P. Robillard. “How API Documentation Fails”. In: *IEEE Software* 32 (4 Aug. 2015), pp. 68–75. DOI: 10.1109/MS.2014.80. URL: https://doi.org/10.1109/MS.2014.80.

[164] April Yi Wang, Andrew Head, Ashley Zhang, Steve Oney, and Christopher Brooks. “Colaroid: A Literate Programming Approach for Authoring Explorable Multi-Stage Tutorials”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI ’23. DOI: 10.1145/3544548.3581525. URL: https://doi.org/10.1145/3544548.3581525.

[165] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. “How Data Scientists Use Computational Notebooks for Real-Time Collaboration”. In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (2019). DOI: 10.1145/3359141. URL: https://doi.org/10.1145/3359141.

[166] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. “Callisto: Capturing the "Why" by Connecting Conversations with Computational Narratives”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376740. URL: https://doi.org/10.1145/3313831.3376740.

[167] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. “A large-scale empirical study on code-comment inconsistencies”. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 53–64.

[168] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. “Chronicler: Interactive Exploration of Source Code History”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI ’16. San Jose, California, USA: Association for Computing Machinery, 2016, 3522–3532. ISBN: 9781450333627. DOI: 10.1145/2858036.2858442. URL: https://doi.org/10.1145/2858036.2858442.

[169] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. “How do developers utilize source code from stack overflow?” In: *Empirical Software Engineering* 24 (2019), pp. 637–673.

[170] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. “What do developers search for on the web?” In: *Empirical Software Engineering* 22.6 (Dec. 2017), pp. 3149–3185. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9514-4. URL: https://doi.org/10.1007/s10664-017-9514-4.

[171] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. “Understanding Feature Evolution in a Family of Product Variants”. In: *2010 17th Working Conference on Reverse Engineering*. 2010, pp. 109–118. DOI: 10.1109/WCRE.2010.20.

[172] Young Seok Yoon and Brad A. Myers. “A longitudinal study of programmers’ backtracking”. In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014, pp. 101–108. DOI: 10.1109/VLHCC.2014.6883030.

[173] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. “Example Overflow: Using social media for code recommendation”. In: *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 2012, pp. 38–42. DOI: 10.1109/RSSE.2012.6233407.

[174] Iyad Zayour and Timothy C Lethbridge. “A cognitive and user centric based approach for reverse engineering tool design”. In: *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. 2000, p. 16.

[175] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, Ying Zou, and Ahmed E. Hassan. “An Empirical Study of Obsolete Answers on Stack Overflow”. In: *IEEE Transactions on Software Engineering* 47.4 (2021), pp. 850–862. DOI: 10.1109/TSE.2019.2906315.

[176] J. Zhi, V. Garousi-Yusifoglu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe. “Cost benefits and quality of software development documentation: a systematic mapping”. In: *Journal of Systems and Software* 99 (2015).