Using Annotations for Sensemaking About Code

Amber Horvath
ahorvath@cs.cmu.edu
Human-Computer Interaction Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Brad A. Myers
bam@cs.cmu.edu
Human-Computer Interaction Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Andrew Macvean
amacvean@google.com
Google
Seattle, Washington, USA

Imtiaz Rahman
imtiaznyc1@gmail.com
Hunter College
New York City, New York, USA

ABSTRACT

Developers spend significant amounts of time finding, relating, navigating, and, more broadly, making sense of code. While sensemaking, developers must keep track of many pieces of information including the objectives of their task, the code locations of interest, their questions and hypotheses about the behavior of the code, and more. Despite this process being such an integral aspect of software development, there is little tooling support for externalizing and keeping track of developers' information, which led us to develop Catseye – an annotation tool for lightweight notetaking about code. Catseye has advantages over traditional methods of externalizing code-related information, such as commenting, in that the annotations retain the original context of the code while not actually modifying the underlying source code, they can support richer interactions such as lightweight versioning, and they can be used as navigational aids. In our investigation of developers' notetaking processes using Catseye, we found developers were able to successfully use annotations to support their code sensemaking when completing a debugging task.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software evolution.*

KEYWORDS

Annotations, notetaking, software engineering, sensemaking, code comprehension

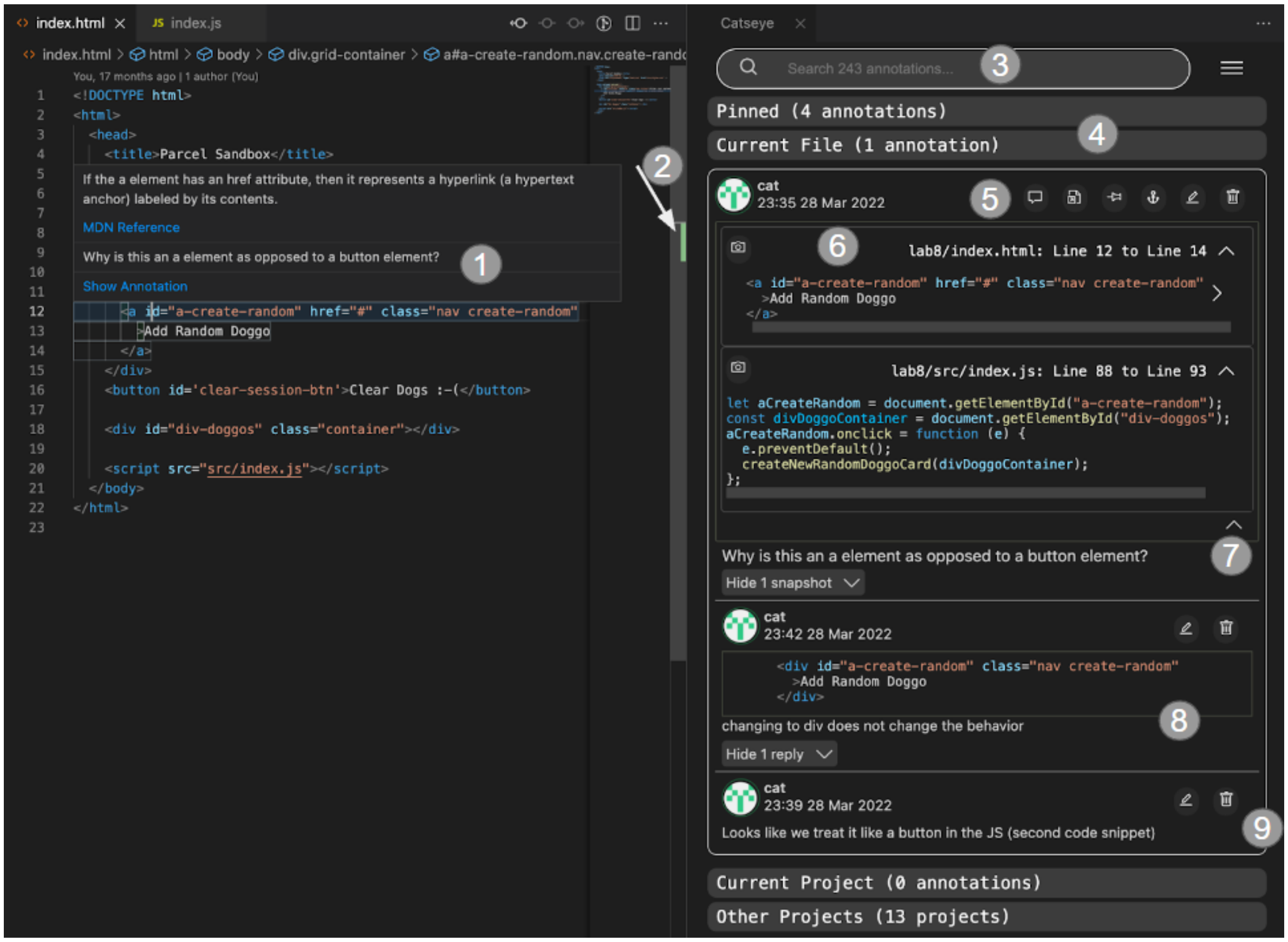
ACM Reference Format:
Amber Horvath, Brad A. Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3526113.3545667

1 INTRODUCTION

Modern software engineering requires developers to make sense of large amounts of unfamiliar code [51]. Whether this is in the context of using an application programming interface (API) or other software library in ones' own code [40], contributing to an open source project for the first time [52], or maintaining and debugging a large project [30], all of these different situations require a developer to locate one or more patches of code, comprehend the code, such that they can debug or modify the code to their own needs, then actually make those modifications [28, 30, 39]. Even if the change and the code base are relatively small, these tasks are still cognitively demanding. The developer must maintain their own task context [39], while also keeping track of the various questions [50] and hypotheses [60] they have about the code, the answers they find to these questions [32, 50], their code locations of interest (commonly referred to as the "working set" [5, 10, 28, 63]), the different versions of the code they try [24, 62], and how those various versions produce differing outputs [24]. Keeping track of all of this information becomes even more difficult considering that programming tasks often span multiple days, and a developer may be working on multiple programming tasks at the same time [42]. Even shorter, simpler tasks can suffer from interruptions, resulting in the developer spending time and energy trying to regain their original task context [15, 41, 43].

With so much information to keep track of, developers employ a variety of techniques to try and externalize this information with varying degrees of success. For example, a common tactic when leaving a programming task is to use environmental cues such as an open file to remind the developers of what they were working on [28, 41, 43]. While these cues can reduce resumption time after an interruption [43], they often suffer from a lack of sufficient context to fully jog the developers' memory. Other mechanisms for keeping track of information include code comments and, more broadly, *notetaking*.

Code comments are commonly utilized for keeping track of open tasks [39, 58] and can be used as navigational aids [54, 58], but are not commonly used for keeping track of the other previously-mentioned information needs developers have such as facts learned or open questions. This may be partially because the cost of externalizing this information, especially when the information may be incorrect, is too high [43], and these code comments must then be cleaned up [53]. Further, there are situations in which developers


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)
MDN Reference
Show Annotation
Catseye
Search 243 annotations...
Hide 1 snapshot
Hide 1 reply
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.

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

```
        <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?

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

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 1: Catseye as it appears in Visual Studio Code shown with annotations on a sample HTML file. (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 informational pop-up underneath 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 corresponding annotation in the pane. (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, save for pinned annotations which appear in their own section at the top. In this case, the "Current File" section is open and expanded while the other sections are collapsed. (5) is an annotation – the top of the annotation shows the author and creation time information on the left, and the buttons corresponding to what operations can be performed on the annotation on the right. (6) shows the two code anchors for the annotation, with the top anchor in the current file, while the second anchor is in a different JavaScript file. (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, further contextualizing and building upon their original thought, given what they discovered in the JavaScript file.

are reticent or not even allowed to add comments to code, since that involves editing the source. Reasons for not editing the source include not having proper credentials to edit the code (e.g., for an external library or API code), not wanting all of their collaborators to see their comments [53], and not wanting to take ownership of the code or check into version control any changes that just involve comments. Given these issues of editing the source, notetaking outside of the editor is commonly used as a way of externalizing some of this information [41] with some developers reporting that they use their notes not only for task-tracking, but also for comprehending code and later relaying their insights to other developers [37]. This suggests that there is a need for a more lightweight mechanism for keeping track of code-related information that is not typically externalized in code comments, but would benefit from utilizing the context of the source code.

In order to support this notetaking need more directly, we developed an annotation system, Catseye (see Figure 1). In our design, we aim to support experienced developers who are performing large-scale, complex code understanding activities (e.g., balancing different tasks, reading through lots of unfamiliar code, and performing maintenance on the code base), and we designed annotating features to support those needs. We chose to use annotations since they contextualize the note to the code, without having the note's content be *in* the code base. Annotations have this advantage over traditional code comments, along with other benefits: annotations can be put on any text (while code comments can only be put in specific places), annotations can be attached to a range of code while code comments are localized to one specific place, annotations use the same syntax everywhere while code comments use different syntax in different programming languages (with some languages such as JSON not having comments at all), annotations can have threaded conversations while code comments do not directly support conversations, and editing code comments can sometimes result in parsing errors while annotations never change the way the code behaves.

Therefore, annotations can serve as a unified interaction technique for providing contextualized, code-related information that would not or should not be put in code comments, while providing additional utilities that make this information more usable. Catseye provides the following features to support developers when keeping track of information:

- *Ephemeral Annotations* to capture developers' questions, open tasks, hypotheses, and other thoughts about the code, without modifying the source code. We use the term "ephemeral" as a contrast to the more permanent nature of code comments.
- *Permanent Annotations*, since prior work has found that developers' questions about a code base can serve as a useful learning resource [55], Catseye provides annotations that can be saved along with the source files, or turned into regular code comments.
- *Replies* where developers can follow-up on the content of their annotations with additional insights, answers to questions, output generated by the system, or other related information.
- *Multiple code anchors* to support developers' navigation of the code through creating working sets of code patches of interest.
- *Pinned annotations* so developers can cluster, prioritize and more seamlessly navigate the code base.
- *Code snapshots* so developers can capture versions of their code without needing to use more heavyweight mechanisms such as version control systems, which developers have been reticent to use when doing "exploratory programming" [24].
- *Capturing system output*, stack traces, error messages, and other textual information useful for sensemaking, through support for copy-pasting the output into the annotation or associating the output with a specific version of the code.
- *Search* so developers can easily find their notes of interest.

To evaluate Catseye's support for sensemaking about code, we ran a small user study. Developers needed to keep track of information that we expect Catseye to help with while attempting to debug some purposefully confusing code. We investigated how developers created and utilized annotations when comprehending unfamiliar and complex code (experimental condition), and how these annotations compared to traditional developer notes (control condition). We found that developers were better able to debug when using Catseye compared to the baseline, and revisited and used their annotations over 2 times more often, on average, than the control condition used their notes.

In this work, we contribute the following:

- A unified mechanism with a collection of features informed by prior work to support developer information needs integrated in our system, Catseye (Section 3).
- A study comparing Catseye with conventional notetaking that demonstrates (Sections 4 and 5):
  - Developers using annotations more often externalize their questions, while traditional notes more often represent open tasks and facts, suggesting annotations are successful in eliciting more ephemeral information.
  - Developers using annotations more often revisit their annotations in comparison to regular notes, and use their annotations in different ways, such as for navigating their code and for resolving their open tasks.

2 RELATED WORK

Our research builds upon prior work that seeks to understand what causes developer confusion and how developers externalize this information, and systems that have tried to help. We also discuss other annotation systems.

2.1 Code Comprehension

Prior literature has extensively studied how developers attempt to learn unfamiliar code [11, 31, 37, 47]. A frequent theme that arises is that understanding unfamiliar code is very cognitively demanding, as developers are attempting to keep track of not only the objectives of their task, but also their developing mental model of how the code is structured [30, 50], the data flow of the program [50], their "working set" of code patches [28], information they have learned [32], and so on. Sillito et al. report that various developer information tracking tasks, such as composing code patches across different files, are difficult and have no support in traditional IDEs [50]. Our work seeks to alleviate some of this cognitive load through directly supporting developers' lightweight notetaking while retaining the context that led to the developers' confusion through annotating, which has been effective in other developer contexts [22].

2.2 Developers' Commenting and Notetaking Behaviors

Code comments have been the subject of prior work that seeks to understand how and what information developers choose to document through comments [20, 48, 53], whether these comments are actually useful [4, 44, 61], and how these comments are later used [14, 45] and cleaned up [56, 57]. 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 [48]. Other work has also found a need for code comment clean up, such as Storey et al.'s work which found that developers commonly leave code comments for bug tracking, but that these comments are sometimes left unattended and can decrease code comprehension [53]. With our system, ephemeral annotations can all be deleted together, and never pollute the real code.

Developer notetaking is less extensively studied than code commenting, but research does provide evidence that developers do take notes and can benefit from them. Prior work found that 77% of developers use notes to keep track of their progress during programming tasks and 75% of these notes are unsituated, meaning they are not kept in the editor [41]. Other studies found that developers take notes for their own comprehension [37, 41], for keeping track of useful resources [34], and for keeping track of their progress on a development task [36, 41]. They also sometimes use these notes as a resource when sharing information with other developers [37]. However, these notes suffer from being non-contextualized, which makes them hard to reuse and difficult to understand if they are not used immediately [37, 43], and these notes have a tendency to get lost [34]. We extend this notetaking work and code commenting research through our tool that makes notetaking for developers more beneficial and our study which reports on how developers currently take notes and the content of these notes.

2.3 Tools to Support Developers' Information Tracking Needs

Tooling support for code comprehension has been an open research challenge given that developers have their own workflows and their own idiosyncratic practices. Some work has attempted to augment current practices through creating more advanced commenting mechanisms, such as TagSEA [54, 58] and other code bookmark projects [18], where, given specific formatting of a source code comment, the IDE will label that code and make it a navigational way point. These tools fit into developers' established work practices but do not resolve the inherent issue of cluttering the code with unnecessary information. Other tools have specifically supported managing multiple code versions and their output [19, 24, 25], managing developer online learning resources [6, 22, 34, 35], and supporting code navigation [5, 10]. Code navigation, in particular, has been shown to take up 35% of developers working time when performing maintenance tasks [30]. Researchers have created tools that attempt to make code navigation easier by understanding the user's task and the relevant code fragments [5, 10], and predict developers' navigation paths to support debugging [33]. Our work attempts to support all of these activities through a lightweight annotating metaphor where the annotations support versions and output, can serve as navigational aids, and can hold references to developers' various learning resources.

Some prior tools have also focused on facilitating developers' explorations of their implicit questions about their code in-situ. Unravel allows data scientists to view summaries of each step of their fluent R code, and re-run each step within the series of function calls [49]. Henley et al. developed a prototype "inquisitive code editor" system that flags code that may confuse novices and quizzes the novices on the expected behavior of those code snippets [21]. These tools show the efficacy of some of Catseye's design choices through succeeding in supporting in-context question-asking and answering, but differ from our design in that we allow the developer to formulate and write out their own questions. Further, these tools support specific populations of programmers who deal with specific issues – data scientists, who need assistance in managing and understanding their data's changing shape through code execution, and novices, who are still learning basic programming concepts. Our tool differs by supporting more experienced developers who are performing large-scale information tracking, where the developer would benefit from externalizing this information while leveraging the context of the source code.

2.4 Annotation Tools

Annotation tools have been developed to help with active reading [16], the act of taking notes while reading. Annotating has been shown to help with sensemaking by allowing the annotator to offload some of their thoughts when processing complex information or comprehending confusing materials [26, 27]. Given these benefits, annotating is supported in applications designed to support reading (e.g., Adobe Acrobat comments) and writing (e.g., Google Doc and Microsoft Word comments). Annotation tools also exist as internet browser plugins, such as Hypothes.is [23] and our Adamite tool [22], they have been used successfully for learning in classrooms [64], and have served as integrated question-and-answer forums for websites [9, 59].

Most relevant to our work are Codepourri [17], a system that allows developers to annotate Python code execution steps to generate tutorials for newcomers to programming, Carter's CodeTour [7], a Visual Studio Code extension which allows developers to create guided walkthroughs of their code base through annotating and linking lines of code, and the Synectic IDE [1], which supports linking and annotating code files. All of these systems showed success, with developers creating annotations that are useful for *others*, but did not investigate how annotations may be useful for the *original author*, despite this being a core benefit of annotation systems. Further, neither Synectic nor Codepourri were integrated into a normal programming environment, leaving annotating impossible during regular programming tasks.

3 OVERVIEW OF CATSEYE

3.1 Catseye

We developed Catseye– an extension for the Visual Studio Code editor [38] – 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 1). We chose to adopt some of the features of our earlier Adamite system [22] for Catseye, given Adamite's focus on supporting developers' information tracking on the web through annotations. Adamite showed 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.

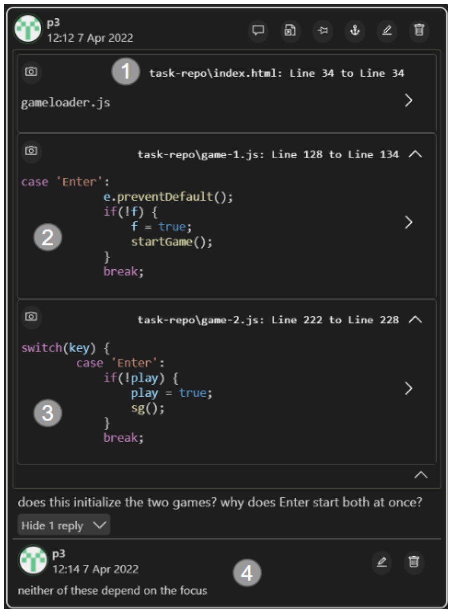
To create an annotation, a developer selects a snippet of code in the editor and, using a keyboard shortcut, the context menu, or Visual Studio Code's Command Palette, indicates that they want to create an annotation. The Catseye pane will update with a preview of the annotation, where the developer 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 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 2 at 2) is updated whenever the developer saves their code (since updating the pane with a new version of their code on every key press was too computationally expensive).

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 1 at 6 and Figure 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 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 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 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.


p3
12:12 7 Apr 2022

1 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;
```

2

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

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

3

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

p3
12:14 7 Apr 2022
4
neither of these depend on the focus

Figure 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.

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 first author's personal usage of the tool (see Section 7). We plan to explore these choices further in future work.

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.

Although this paper focuses more on ephemeral annotation use, we also expect that some of the information that developers choose to annotate may be worth keeping around. A single command will convert an annotation to be a regular code comment, so the information can be easily persisted in the code base, if desired.

Annotations can also be exported to a JSON file so that they may be tracked in version control.

3.2 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. [50] 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 [32]. 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 [41]. 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 [5, 10, 28], 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 [28]. 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 [24], and switching between these versions can be difficult if the prior versions are not retained, especially since they may be inaccessible through undo commands [62]. 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 [24] 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). However, we expect that annotations will be beneficial for other purposes, which we discuss in Section 9.

3.3 Implementation Notes

Catseye is a Visual Studio Code extension which utilizes TypeScript and the Visual Studio Code API for the core logic of the extension, React 17.02 [13] for the user interface, and Google Firestore [12] for storing user profile and annotation data.

When Visual Studio Code is launched, Catseye authenticates the user using both Visual Studio Code and Firestore. The Visual Studio Code API's authentication service authenticates using the user's GitHub account and the GitHub OAuth data is sent to Firestore in order to connect to our database. Once the user has been authenticated, all of their annotations which have not been deleted are pulled into the system. [1] If the user is not authenticated or does not have a GitHub account, they can still use Catseye, but, instead of having their annotations stored on the database, their annotations will be stored locally in a JSON file that the system produces.

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 [46], we leave more robust methods for fixing up broken anchors to future work, along with investigating the

[1] Deleted annotations are kept on the server for recovery and research purposes.

design space of how annotations should be updated or archived given these large changes.

4 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 [29], since these are problem-solving tasks that you can only do once, creating 2 tasks which are independent but of equal difficulty is challenging. All study materials are available in the Supplemental Materials and the study was approved by our institution's Institutional Review Board.

4.1 Method

4.1.1 Training Task. Both conditions included a training task using a repository of website templates[2] 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.

4.1.2 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 see 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. During the full 45 minutes, participants were encouraged but not required to keep track of information in the way they naturally would (control) or to use Catseye (experimental). By making use of the tool (experimental) or other resources (both) entirely optional, we aimed to make the task more realistic and see what participants wanted to use.

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 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 [41, 43].

The code was specifically designed to be confusing in order to make keeping track of information particularly important (see Table 2). Given prior literature around what makes code confusing [50], 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 [3] but that they do not encounter code like this very frequently [4].

4.2 Participants

We recruited 13 participants (5 women and 8 men) using study recruitment channels at our institution, and advertisements about the study on Twitter, Facebook, and LinkedIn. 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.

All study sessions were completed remotely using video conferencing software. Each participant was given access to the GitHub repository with the code used for the main task. Participants were

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

[3] average = 3.4 out of 5, using a 1-to-5-point Likert scale from very dissimilar to very similar

[4] average = 2.6 out of 5, using a 1-to-5-point Likert scale from never to very frequently

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

compensated $25 for their time, save for 1 participant who elected not to be compensated.

4.3 Analysis

Across both conditions, we objectively coded what bugs the participant succeeded in fixing (see Table 1). In the experimental condition, we analyzed the video recordings and log data to count how many annotations each participant authored and how often they interacted with their annotations (including creating replies, additional anchors, and snapshots, pinning an annotation, reading an annotation, editing or deleting an annotation, and navigating using the anchor(s)) 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. [5] 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.

5 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.

5.1 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

[5]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".

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 [50]
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 2: How the study task encapsulates the types of information Catseye supports.

the majority of annotations attached to one line or less of code (71/84).

Across those 84 annotations, developers had a variety of types of information they chose to keep track of through annotations (see Figure 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. Two participants chose not to make any annotations related to their open tasks. Task annotations were followed-up on with replies twice, with one participant adding a reply hypothesizing about why the change she attempted did not work, while another participant added some pseudo-code to her annotation about how she planned to implement her change. 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.

Two annotations were made that did not fit into any of the other categories – one annotation was a message to future readers of the code stating that they are better off rewriting the whole project, while the other was a link to Mozilla documentation. These annotations show other usages for annotations, such as communicating with collaborators and keeping track of useful references that are related to the code.

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,

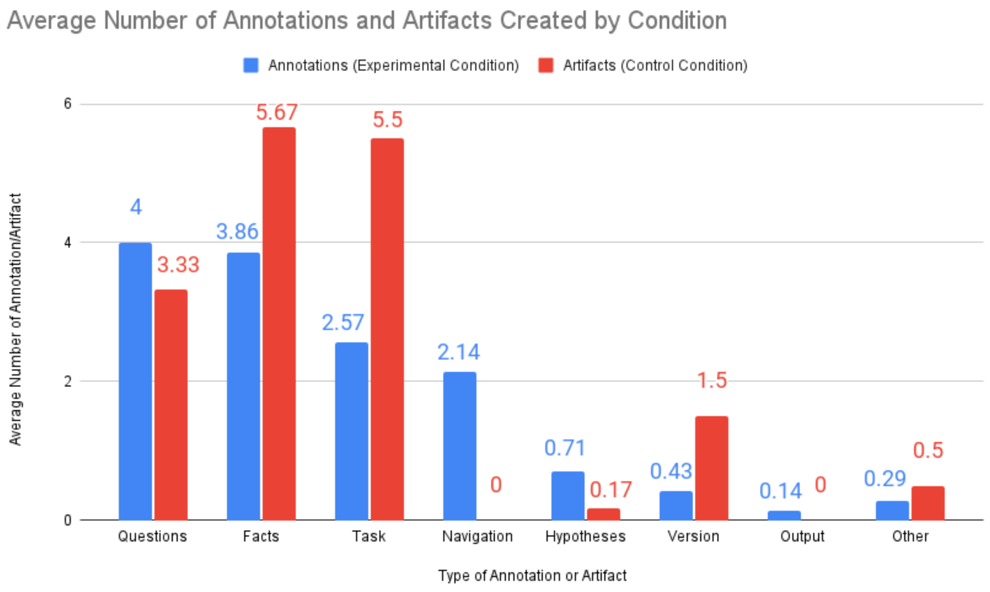


Figure 3: The average number of annotations and artifacts participants created during the study.

all of the control participants had some notetaking 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[6]. 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 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. 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.

Another subtle difference between the control participants' artifacts and the annotations created by the experimental participants is the prevalence of task artifacts (see Figure 3). Control participants created nearly double the amount of task artifacts, on average, and 61% of these task artifacts were code comments that included the text "TODO". In contrast, only 22.2% of task annotations included the text "TODO". Some of the control artifacts also included non-code related tasks, such as asking their "teammates" about certain design decisions, which the experimental condition participants did not create. These "TODO" code comments are similar to the ones described in [53] and may be better supported by Catseye through supporting tagging tasks for better filtering and alerting collaborators if an annotation is made that requires their attention.

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

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

Fact and question annotations were followed up on during the course of the main task, with facts being revisited, on average 2.5 times throughout the task and questions revisited 1.9 times, as participants reminded themselves of important details about the game implementations and found answers to their questions. In contrast, the control participants only revisited their fact artifacts, on average, 0.24 times per session, and their questions, 0.2 times.

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.

Three control participants used their code comments as templates by copy-pasting the comment. These 3 participants copy-pasted 4 code comments and subsequently edited 3 of the 4 pasted comments in order to make a new comment. All of these code comments were "fact" comments that served to document some confusing behavior. This supports the idea that Catseye should allow for copy-pasting *some* annotations when copy-pasting the code the annotation is attached to, such as annotations documenting code behavior, which is discussed more in Section 7.

The control condition averaged 2.33 edits per participant while the experimental participants averaged only .28 edits. Control participants sometimes edited one "main" artifact they created to keep track of bugs they found (6 out of 14 edits across 3 artifacts) and these edits on a "main" note only occurred on external notes. They also edited their notes to add to or clarify their initial facts or questions about the code behavior (5 out of 14 edits across 5 artifacts). These usages are similar to how experimental participants typically used the reply feature with their annotations in order to follow-up on the content, while not editing their annotations.

5.3 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 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.

6 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. 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.

Participants reacted favorably to Catseye while also envisioning improvements. In the post-task survey, participants commented on how they found the commenting system intuitive, enjoyed the anchoring system as a way of connecting parts of the code, liked following up on annotations through replies, and stated that they would find the system useful for their daily work[7]. Participants also commented on a desire to collaborate with their teammates

[7]"I would consider Catseye useful for my daily work", average score 6.16 out of 7, with 7 being "Strongly Agree"

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 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.

using Catseye, they wanted a high-level overview of their annotating behavior and the annotations they have authored, and they requested a way to make higher-level comments on whole files or their general task. Since running the study, we have implemented file-level annotations. Future work should explore how to create high-level summaries of developers' activities using their annotations to further support task tracking, and how to support collaboration through improved GitHub integration.

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.

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 ran. We further plan to allow captured output to not only be text data, but also rich media such as screenshots or videos, which would help for debugging systems like the one used in our study, where the primary output is visual.

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[8]. Prior work has noted that annotations' flexible nature and structure allows them to be used in a variety of ways [2, 3] – 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

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

"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 [8].

Annotations with multiple anchors brought together patches of code that were related to the developers "working set" of interest. For example, P3 wanted to resolve the bug "Unable to Play Games Independently" – he found the "gameloader.js" file and annotated the file, wondering whether this is the source of the issue (see Figure 2). Later, he found that the reason the games were both started is because they both listen to the "Enter" key, so he edited his annotation to reflect that, and added additional anchors pointing to the near identical event listeners across the two game files. He pinned the annotation to remind himself of this information, such that, when he was ready to edit the files, he could resolve the bug and modify the code so that the elements had to be in focus for the key press to start the game. P4 similarly added multiple anchors in order to bring together function calls and function definitions across multiple files that she was confused about.

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 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!) – which we will research further in the future.

7 FIRST AUTHOR USAGE OF CATSEYE

The first author used Catseye while developing Catseye as a form of "dogfooding." Anecdotally, we report on the annotations created by the first author 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 the first author to help herself, either in the short term (so they were ephemeral) or for when she returned to the code later.

Over 6 months, the first author created 122 "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). 31 annotations had a total of 46 replies, 8 annotations had 18 snapshots, 3 annotations had multiple anchors[9], and 7 annotations were pinned[10]. 70 of these 122 annotations were deleted as the first author finished open tasks and iterated over the code, including removing the code the annotation is anchored to which deletes the annotation.

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 37 of the 122 annotations reporting some open action item the first author 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. The first author also created many question annotations (36 out of 122) – 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

[9]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.
[10]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.

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 the first author 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.

The first author found the tool useful for externalizing and thinking through problems and stopped using 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 and, thus, there was a lot of uncertainty about the design and implementation details – information that the first author did not want to commit to the code base as comments in case the information ended up becoming obsolete which is a common problem with code comments [14, 44, 56, 57].

8 LIMITATIONS AND THREATS TO VALIDITY

Our study is limited by the fact that we cannot directly compare the bugs that participants addressed between conditions, since we chose to allow developers to fix whichever bugs they discovered and were motivated to fix. We feel this design resulted in a more realistic study, considering developers normally are mostly self-governing in terms of choosing what to work on and how to balance their tasks, which also allowed us to better assess how developers choose to use annotations for task tracking.

Another limitation of our study is that we created the code used in the study, as opposed to adopting an already problematic code base. We chose to create the code base in order to ensure that it required developers to keep track of the information we wanted to investigate, but future work may see how Catseye helps when working on real projects. We do have evidence that the code base, despite being artificial, is relatively similar to code our participants have encountered in their time as programmers, even though they (thankfully) do not encounter this kind of code often (see Section 4.1.2).

9 FUTURE WORK

While our lab study provides some evidence that Catseye is useful when performing debugging tasks, we want to extend our work by seeing how Catseye is used in other contexts in the wild with a large-scale field study.[11] Given the difference in content between what was annotated in the lab study and what was created by the author when developing a system (see Section 7), we expect a field study where developers are performing a variety of development tasks will further elucidate and quantify these differences.

A core design tenant of Catseye is that the annotations should feel ephemeral and lightweight – we want the annotations and their corresponding code anchor points to stay up-to-date and accurate to prevent confusion. To support this design goal, Catseye checks and updates its internal representation of the anchor starting and ending points (i.e., the starting and ending lines and offsets) for each code anchor that is in the current file whenever that file is updated. This method works well when a developer is working on a project on their own, but does not work for capturing edits made outside that user's instance of Visual Studio Code. Catseye utilizes the GitHub API so it is feasible to use this API to associate each anchor point with a specific commit of the repository, such that, if a new branch is pulled in or checked out, the corresponding annotations can either be updated (in the case of a pull) or archived (in the case of a branch checkout).

Better GitHub integration also allows us to explore how annotations may help developers when *collaborating*. We hypothesize that annotations may be useful in a collaborative setting for better supporting code review and, more broadly, conversations about code. Both of these use cases benefit from an ability to leverage the source code for in-context communication about its content, while not cluttering the code base with these meta-level conversations. We leave to future work more directly supporting collaborative software engineering and exploring the design space of how code annotations should function when developers may have differing versions of the same code across their machines.

10 CONCLUSION

Developers must keep track of and understand many types of information when completing any programming task. In our development of Catseye, we sought to help developers keep track of ephemeral information through a custom-designed annotation system. In our evaluation of Catseye, we found evidence that it supports developers in tracking different types of information, with participants fixing more bugs when using the tool, and annotations differ from traditional notes in terms of the content that developers choose to externalize and in terms of how often that content is revisited. We showcase Catseye as an example of the types of rich interactions which annotation systems can support for developers and how supporting developer sensemaking activities through annotations is a research area that warrants more investigation. We are also optimistic that similar lightweight annotations can be useful to support a wide variety of sensemaking tasks.

ACKNOWLEDGMENTS

This research was funded in part by the NSF under grant CCF-2007482 and by gifts from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors. In addition, we would like to thank River Hendriksen, Michael Xieyang Liu, Emma Paterson, Shannon Bonet, Matthew Shu, our participants, and our reviewers for their thoughtful input and feedback.

[11]Catseye is now available for download at https://adamite.netlify.app/

REFERENCES

[1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. 2020. 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, 1–10.

[2] Maristella Agosti, Giorgetta Bonfiglio-Dosio, and Nicola Ferro. 2007. A historical and contemporary study on annotations to derive key features for systems design. *International Journal on Digital Libraries* 8, 1 (2007), 1–19.

[3] Michael Bernstein, Max Van Kleek, David Karger, and MC Schraefel. 2008. Information scraps: How and why information eludes our personal information management tools. *ACM Transactions on Information Systems (TOIS)* 26, 4 (2008), 1–46.

[4] Jürgen Börstler and Barbara Paech. 2016. The role of method chains and comments in software readability and comprehension—An experiment. *IEEE Transactions on Software Engineering* 42, 9 (2016), 886–898.

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

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *CHI '09* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[7] Jonathan Carter. 2020. *CodeTour*. Microsoft. Retrieved July 1, 2022 from https://marketplace.visualstudio.com/items?itemName=vsls-contrib.codetour

[8] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science* 13, 2 (1989), 145–182.

[9] Parmit K. Chilana, Amy Ko, and James O. Wobbrock. 2012. LemonAid: selection-based crowdsourced contextual help for web applications. In *CHI 2012*. ACM, New York City, NY, USA, 1549–1558. https://doi.org/10.1145/2207676.2208620

[10] Michael J. Coblenz, Amy J. Ko, and Brad A. Myers. 2006. JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange* (Portland, Oregon, USA) *(eclipse '06)*. Association for Computing Machinery, New York, NY, USA, 65–69. https://doi.org/10.1145/1188835.1188849

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

[12] Google Developers. 2022. *Cloud Firestore: Store and sync app data at global scale*. Google LLC. Retrieved March 27, 2022 from https://firebase.google.com/products/firestore

[13] Facebook. 2022. React - A JavaScript library for building user interfaces. https://reactjs.org/

[14] Beat Fluri, Michael Wursch, and Harald C Gall. 2007. 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, 70–79.

[15] James Fogarty, Amy J. Ko, Htet Htet Aung, Elspeth Golden, Karen P. Tang, and Scott E. Hudson. 2005. Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Portland, Oregon, USA) *(CHI '05)*. Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/1054972.1055018

[16] Ingo Frommholz, Holger Brocks, Ulrich Thiel, Erich Neuhold, Luigi Iannone, Giovanni Semeraro, Margherita Berardi, and Michelangelo Ceci. 2003. Document-centered collaboration for scholars in the humanities–the COLLATE system. In *International conference on theory and practice of digital libraries*. Springer, 434–445.

[17] Mitchell Gordon and Philip J. Guo. 2015. Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 13–21. https://doi.org/10.1109/VLHCC.2015.7357193

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

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

[20] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When not to comment: questions and tradeoffs with api documentation for c++ projects. In *Proceedings of the 40th International Conference on Software Engineering*. 643–653.

[21] Austin Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. An inquisitive code editor for addressing novice programmers' misconceptions of program behavior. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 165–170.

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

[23] Hypothes.is. 2012. *Hypothes.is: Annotate the web, with anyone, anywhere*. Hypothes.is. Retrieved March 30, 2022 from https://web.hypothes.is/

[24] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[25] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300322

[26] David Kirsh. 2010. Thinking with external representations. *AI & society* 25, 4 (2010), 441–454.

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

[28] Amy J. Ko, Htet Aung, and Brad A. Myers. 2005. 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* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 126–135. https://doi.org/10.1145/1062455.1062492

[29] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[30] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

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

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

[33] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215. https://doi.org/10.1109/TSE.2010.111

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

[35] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. 2022. 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* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491102.3501968

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

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

[38] Microsoft. 2022. *Visual Studio Code*. Microsoft. Retrieved March 28, 2022 from https://code.visualstudio.com/

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

[40] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (2016), 62–69. https://doi.org/10.1145/2896587

[41] Chris Parnin and Robert DeLine. 2010. *Evaluating Cues for Resuming Interrupted Programming Tasks.* Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/1753326.1753342

[42] C. Parnin and C. Gorg. 2006. Building Usage Contexts During Program Comprehension. In *14th IEEE International Conference on Program Comprehension (ICPC'06).* 13–22. https://doi.org/10.1109/ICPC.2006.14

[43] Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Software Quality Journal* 19, 1 (2011), 5–34.

[44] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 91–100. https://doi.org/10.1109/ICSME.2014.31

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

[46] Steven P. Reiss. 2008. Tracking Source Locations. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08).* Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1368088.1368091

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

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

[49] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology.* 198–207.

[50] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451. https://doi.org/10.1109/TSE.2008.26

[51] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers.* 174–188.

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

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

[54] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. 2009. How Software Developers Use Tagging to Support Reminding and Refinding. *IEEE Transactions on Software Engineering* 35, 4 (2009), 470–483. https://doi.org/10.1109/TSE.2009.15

[55] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. 2012. Code Comprehension Problems as Learning Events. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12).* Association for Computing Machinery, New York, NY, USA, 81–86. https://doi.org/10.1145/2325296.2325319

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

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

[58] Christoph Treude and Margaret-Anne Storey. 2012. Work Item Tagging: Communicating Concerns in Collaborative Software Development. *IEEE Transactions on Software Engineering* 38, 1 (2012), 19–34. https://doi.org/10.1109/TSE.2010.91

[59] Laton Vermette, Shruti Dembla, April Y. Wang, Joanna McGrenere, and Parmit K. Chilana. 2017. Social CheatSheet: An Interactive Community-Curated Information Overlay for Web Applications. *Proc. ACM Hum.-Comput. Interact.* 1, CSCW, Article 102 (Dec. 2017), 19 pages. https://doi.org/10.1145/3134737

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

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

[62] Young Seok Yoon and Brad A. Myers. 2014. A longitudinal study of programmers' backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* 101–108. https://doi.org/10.1109/VLHCC.2014.6883030

[63] Iyad Zayour and Timothy C Lethbridge. 2000. 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.* 16.

[64] Sacha Zyto, David Karger, Mark Ackerman, and Sanjoy Mahajan. 2012. Successful classroom deployment of a social document annotation system. In *CHI 2012.* ACM, New York, NY, USA, 1883–1892. https://doi.org/10.1145/2207676.2208326