Highlights

A Sensemaking Analysis of API Learning using React

Caitlin Kelleher, Michelle Brachman

- Interprets and formalizes sensemaking within the context of API Learning.
- Describes an API learning process of the React API via sensemaking and identifies ineficiencies in the process.

A Sensemaking Analysis of API Learning using React

Caitlin Kelleher[a], Michelle Brachman[b]

[a]Washington University in St. Louis, 1 Brookings Dr., St. Louis, 63130, Missouri, United States
[b]IBM Research, 314 Main St., Cambridge, 02142, Massachusetts, United States

Abstract

Current programming practices rely heavily on the use of APIs (Application Programming Interfaces) and frameworks. However, APIs can be challenging to learn and use. Existing research focuses on specific barriers programmers encounter while learning APIs, providing a fragmented understanding of the process. In this paper, we analyze the holistic process of twelve programmers learning the React JS API using sensemaking theory as a guiding framework for qualitative coding of behaviors. We describe how these API learners moved through sensemaking stages and how they interacted with information during each sensemaking stage. Our results highlighted programmers' tendency to seek understanding when they encountered problems.

1. Introduction

Programmers of all skill levels often need to learn APIs (Application Programming Interfaces) and frameworks for modern software development, as the number of new and updated APIs rapidly grows [1, 2]. Programmers rarely learn how to use APIs through formal education. Instead, programmers tend to learn APIs on-the-fly to meet the needs of a given project [3]. Consequently, programmers must find and use web resources to both understand the capabilities and framework of an API as well as how to apply that API to their problem context. Researchers have documented the trials and tribulations of programmers attempting to learn and use APIs on their own [4, 5]. While helpful, this research often focuses on particular barriers, such as finding and understanding individual pieces of information [6] or integrating individual example code snippets [7], leaving us without a cohesive picture of the API learning process overall.

Preprint submitted to Journal of Computer Languages October 3, 2023

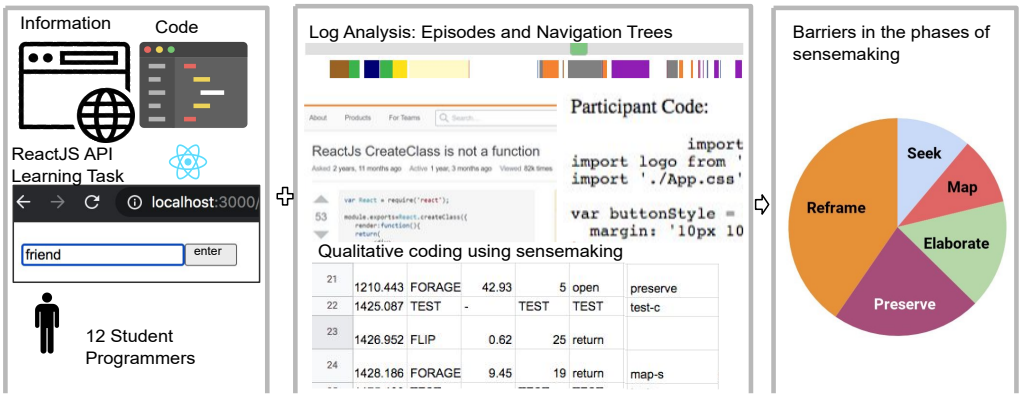

Figure 1: Overview figure: We ran a study with 12 student programmers and used a log analysis method combined with qualitative coding to better understand barriers in API learning through the lens of sensemaking.

To better understand the use and integration of information sources during API learning, we applied a sensemaking lens. Sensemaking models describe how people seek out information, integrate information, form hypotheses about a previously unknown topic, and attempt to assess their correctness [8, 9, 10]. In contrast to theories like information foraging [11], which can describe parts of the API learning process, sensemaking can be used to understand the complete process. Sensemaking provides a way to explore the obstacles that people encounter and their process in addressing those obstacles. Rather than considering each information search independently, sensemaking enables us to consider how different information searches relate to the programmer's goals and holistic task process. Understanding what kinds of information programmers seek to use and the circumstances under which it is sought can help us to design better documentation and API learning support tools. By better articulating the kinds of questions have and the circumstances under which they occur, we can design learning supports to meet programmers where they are.

This paper asks two core questions:

1. How do programmers learning an API move through the sensemaking process?
2. Where in the sensemaking process does API learning break down?

To answer these questions, we analyzed data from a study in which twelve

student programmers with little or no experience attempted to complete a programming task using the React web API 1 in one hour. We grouped programmers' actions into sensemaking episodes. Study participants found relevant information relatively easily but struggled to effectively use that information. In particular, programmers frequently replaced their starting code and removed code additions. Code removals and replacements did not contribute towards task completion and the activities had poor supports for learning.

This work makes two main contributions:

1. An interpretation and formalization of sensemaking within the specific context of API learning.
2. A description of the API learning process via sensemaking that identifies ineficiencies in the process.

2. Related Work

This paper builds on prior research in three areas: API learnability, information seeking while programming, and analysis of logs for understanding user behavior.

2.1. API Learnability

Prior research in API learnability has three main themes: API learning barriers, API design for learnability, and documentation.

API Learning Barriers: Research suggests that the common API learning barriers programmers encounter include: 1) framing appropriate questions [12, 13], 2) combining multiple API elements to solve a problem [4, 5, 12], 3) extracting needed information from documentation [4, 5, 12], and 4) managing collected learning resources [14, 15]. While identifying these problems represents an important step towards understanding API learning, the picture is far from complete. To design effective supports for these problems, we need to understand how these problems arise from the process programmers use. However, much of the research into API learning barriers has relied on self-report data such as surveys [4, 5], interviews [4, 5], and analysis of questions posted online [13]. These techniques, by their nature, cannot capture the learning and development process. In-person studies have focused on questions programmers pose and struggle to answer [12], how API functionality is discovered [16], and the use of external memory throughout

API learning [14]. Our work contributes to API learning barriers research by exploring the process by which programmers collect and transfer information from the search context to the programming context and where that process can break down.

API Design for Learnability: A second group of work explores how to design APIs for learnability with the goal of identifying properties that contribute to learnability. Work in this category takes a variety of approaches including iteratively improving existing APIs [17, 18, 19], analyzing the cognitive properties of APIs [20, 21], and using controlled studies to identify the impacts of specific API design choices [22, 23]. Research into API design for learnability has made important strides towards understanding what makes an API learnable or not. However, not all API designers will follow the resulting guidelines.

Documentation: API learners report that documentation is a critical resource, but that today's API documentation falls short [12, 4, 5]. Research in this area seeks to improve documentation through understanding the properties of useful documentation by conducting usability studies [24], programmer interviews [25], and analyses of existing documentation [26]. Guidelines resulting from these efforts emphasize programmers' preferences for obtaining API information on-demand and the importance of code examples. These results are similar to findings in opportunistic programming [3, 27, 28, 29, 30, 31].

Existing research in API learnability covers a variety of aspects, but it lacks cohesion. By following the information flow from the search context into the programming context, we add to the body of knowledge surrounding how programmers approach learning new APIs and the obstacles they encounter while doing so.

2.2. Information Seeking while Programming

Two ways researchers have found that programmers seek information is during opportunistic programming and by foraging for information.

Opportunistic programming research explores the relationship between information searches and programming behavior. Opportunistic programmers often aim to get code up and running quickly by searching for code-related information on an as-needed basis [3]. Searches are typically motivated by one of three goals: 1) to remind programmers of familiar content, 2) to clarify or extend a known concept, or 3) to learn new concepts. After past-ing copied code into an IDE, opportunistic programmers frequently edit that

code, but reused code is rarely tested [32]. This style of programming is often adopted by those who are programming to achieve a goal [33]. Due to their lack of familiarity with React, participants in our study primarily conducted searches to learn and searches to extend knowledge. This paper contributes an analysis of programmers' processes using a model of sensemaking, and identifies where and how the sensemaking process breaks down.

Information seeking during programming has also been explored through the lens of information foraging theory. Information foraging theory conceptualizes information seekers as predators, seeking information as prey based on scents that help the information seekers better find what they need in larger information patches [34, 35]. Information foraging theory has been used to design tools to support programmers [36, 37, 38], and more relatedly, to classify programming behavior [39, 38, 40, 41]. Researchers developed a novel information foraging model that handles evolving models and captured programmers' information foraging over time and showed that this model could support tool design [42, 43]. However, information foraging models focus on attempting to predict information needs algorithmically, rather than explaining the process and barriers in API learning. Further, much of this work has focused on regular programming practices, rather than programmers learning new APIs. Finally, recent work considered API learning through the lens of information foraging theory [14, 15]. However, information foraging theory is limited in understanding the reasoning behind information forages. In this paper, we capture API learning behaviors through the lens of sensemaking, which supports understanding of the motivation behind information searches.

2.3. Logs for Understanding Behavior

Logs are a popular way to capture user data, especially on the web. New big data analysis methods make user behavior more valuable by providing insight into their behavior [44, 45, 46]. For example, log data can provide critical user information about user behavior that isn't necessarily observable otherwise, such as in the context of social networking sites, where many user interactions are simply browsing, rather than necessarily interacting in observable interactions [47]. Some work also goes beyond capturing the data and performing simple analysis to synthesis and summarization, such as in order to identify suspicious users [48]. Researchers have used several methods to do this, like looking at transitions between windows to generate PC user behavior models [49], as well as by analyzing 'panel' logs, which capture

randomly selected user behavior, and linking behavior with user searches [50].

User logs are used in a variety of ways in HCI research [51], such as understanding user behavior, as well as for comparative analysis. We leveraged existing work in logs for understanding behavior and connected our log behavior with sensemaking stages through qualitative coding.

3. Sensemaking Background

We use sensemaking theory to understand and describe API learners' processes as they work towards creating specific functionality using an unfamiliar API. To date, sensemaking has rarely been incorporated into research understanding programmer behaviors [52] and has not been leveraged to understand API learning behaviors. At their core, sensemaking models aim to describe the process that people go through when trying to interpret, understand, and use complex information. Sensemakers need to recognize which information is relevant, and then organize that relevant information into a structure that can address a particular information need. When the sensemaking process breaks down, it is often because programmers do not know what to do or how to select needed information.

Researchers have proposed a number of sensemaking models for different information-rich contexts: communication processes [53, 9, 54], information extraction [10], and expert decision making [55, 8]. Dervin describes the process of sensemaking through a metaphor in which a person encounters a situation [53, 9, 54]. That person brings with them a history and experiences. They perceive a gap in understanding that may manifest as questions, confusion, or even anxiety. To address that gap, they try to build a bridge using a variety of cognitive tools such as ideas and thoughts and emotional tools such as feelings and intuitions. The result of that bridging process is the outcome of sensemaking. In Russell's model [10], sensemaking is described as an information-extraction activity that accompanies information foraging [10] and can be guided by existing schema that information seekers may have [56]. Klein's model is largely consistent with Russell's, but adds a search for an organizational structure (a frame) and includes the consideration of multiple frames [8].

We chose to base our sensemaking analysis on the Klein model of sensemaking which is grounded in expert decision making in a variety of domains (e.g. fire investigation [8] and medical diagnosis [55, 8]). Sensemaking is

presented as an activity embedded in the physical and social world where sensory experiences play an important role [57, 55]. Sensemaking includes recognizing which sensory experiences are relevant to an explanation, holding and comparing multiple potential explanations based on contextual observations, and determining how best to act by considering contextual knowledge [8]. The Klein model centers around the creation and manipulation of frames, structures that describe the relationships between entities, similar to a schema. Klein describes seven activities that center on frames: mapping data and frame, elaborating a frame, questioning a frame, preserving a frame, comparing frames, re-framing, and constructing or finding a frame. Together these activities allow us to obtain a detailed understanding of the sensemaking process. We describe the alignment of API learning behaviors to Klein's stages of sensemaking in Section 7.

4. Study

In order to understand how programmers move through sensemaking as they learn a new API, we analyzed data from a lab study [58] in which programmers attempted to complete programming tasks with an unfamiliar API. Data from this study has also been analyzed with a model that incoorporated Cognitive Load Theory, Information Foraging Theory, and the use of External Memory in API learning. In this work, we re-examine programmers' learning processes via sensemaking in order to understand how programmers' information gathering and use informs how they make sense of a new API. First we describe our study, followed by our approach to using sensemaking as a lens to understand programmer behavior.

4.1. Participants

We analyzed data from twelve participants. In the original study, there were fourteen participants, with one data set lost due to a technical error and one participant for whom logging did not capture the necessary information for this analysis. Participants were recruited using a mailing list for a Computer Science Department. Eleven of the twelve participants were men and participants' ages ranged from 19 to 34 ($M = 22, SD = 4$). All participants were students and had programming experience but their programming experience varied: four had professional experience, while the others had only programmed for courses. Five participants had no experience with React or JavaScript, five only had JavaScript but no React experience, and two

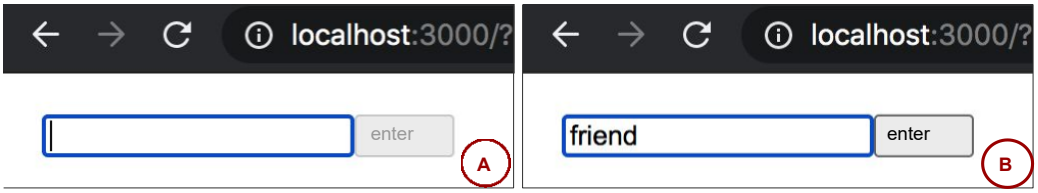

Figure 2: In a completed task, there is an input area and a disabled button, as shown in (A). When 'friend' is typed into the input area, the button becomes enabled, as shown in (B).

had used React before, though not extensively. Participants received a $20 Amazon gift card for participating.

4.2. API Learning Task

We analyze data from an API learning task participants worked on for up to one hour. The programming task involved using ReactJS, which we chose because it is a common but dificult to learn API. Participants were provided with a basic template application. Using this template, participants attempted to create a text input area and a button, as shown in Figure 2. According to the task, the button should be disabled initially and become enabled when a user typed 'friend' into the text area. We chose this task because previous research has shown that connecting components using APIs is particularly challenging [4, 5, 12]. Participants completed this task using the Atom code editor and a web browser.

4.3. Data Collection

We collected and analyzed web pages participants visited, participants' navigation history, and their code changes in the code editor. This data also includes code testing actions, which took the form of navigation to a localhost webpage in the web browser. Through the code changes, we are also able to determine participants' success on the task, marked by four task subgoals.

5. Data Analysis Approach

The goal of our data analysis process was to extract meaningful sequences of user behavior that correspond to sub-goals in their programming task. To do so, we used a three step, bottom-up process:

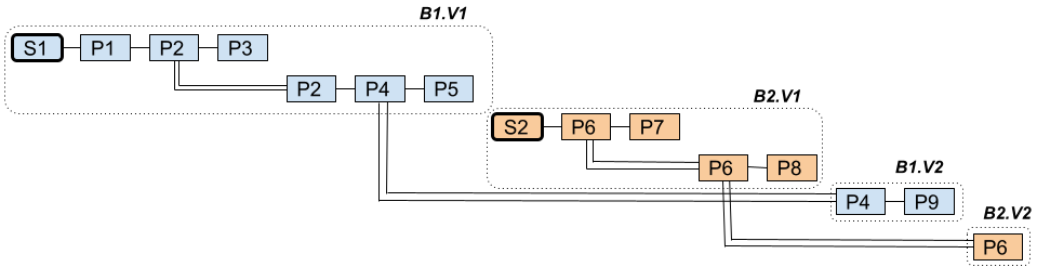

Figure 3: This is a navigation tree of a programmer's web foraging behavior showing two branches that are each visited twice. Nodes (searches or web sites) in the tree are arranged to show the sequence in which the programmer visited them. Each branch visit is outlined in a dotted rectangle. Single lines connecting two nodes indicate that the user followed a link on the first page to reach the second. For example, the programmer clicked on a link in S1 to reach P1 and a link in P1 to reach P2, and so on. In this figure, the first branch (B1) is generated through search S1. The programmer then views pages P1-P3 before backing up to P2 and then navigating to P4 and then P5. The second branch (B2) is generated by the search S2. Based on S2's results, the programmer then views P6 and P7, returns to P6 and then views P8. After viewing P8, the programmer returns to the results of branch B1, resulting in a second branch visit (B1.V2) where the programmer re-views P4 and navigates to P9. Finally, the programmer visits B2 a second time (B2.V2) and re-views P6.

- Step 1: We group web-page visits into sequences that correspond with a single information goal (or branch visits).
- Step 2: We use a qualitative coding process to label each branch visit with a sensemaking stage.
- Step 3: We extract episodes of sensemaking activities (branch visits, coding, and testing) to capture the process by which task progress occurred.

6. Branch Visits: Grouping web activity by information goal

The goal of segmenting web-search activity by information goal is to enable individual analysis of the role of each information seeking activity. This is important because information is gathered for different reasons throughout the development process and contributes to different stages in the sensemaking process. For example, early in the development process, programmers might look for an example on which to build. Later, programmers might use

searches to understand unexpected behavior or figure out how to combine different pieces of functionality.

To segment our web search information by goal, we define two constructs: branches and branch visits. We define a branch of information as all of the pages a programmer reaches while pursuing a single information target. Branches bring together all of the information considered in pursuit of a single information goal. However, branches are sometimes stored and visited multiple times. In response, we additionally define a branch visit as a set of pages belonging to a single branch visited during a continuous time interval (i.e. not interrupted by testing or foraging on a different topic). Note that a programmer may visit a branch multiple times, resulting in multiple branch visits.

To extract branches and branch visits we: 1) generate a navigation tree, 2) identify information branches, and 3) extract branch visits.

Generating Navigation Trees We first construct a tree that captures participants' full navigation histories based on our log information. The navigation tree uses link navigation to define a parent-child relationship. If a programmer on Page A clicks a link to Page B, Page B will appear as a child of Page A in the navigation tree. As the programmer navigates to new pages throughout their programming sessions, we record their page visits as well as the tab and window ownership associated with each using a custom Chrome plugin. We reassemble this information to create a tree showing the origin of each page navigation and their children created through navigation.

Identifying Information Branches: Next, we identify information branches from navigation trees. Information branches are segments of the navigation trees and include: 1) a seed page and 2) all children reached through navigation from this seed page and its children. The seed pages are those pages that the programmer reaches by expressing a new information target. In most cases, seed pages are search pages that include the information target via keyword search terms. In some cases, programmers perform multiple searches before visiting a single result page. We group all of these no-navigation searches together with the final one that does result in navigation as part of one branch. Additionally, a programmer may express a need for information without articulating the intended target by manually typing in an address or by navigating to the React documentation via a link in the template code provided to them. In these non-search cases, we may not know what the programmer's information need is, but the action of opening one of these resources expresses the need for new information. Note that because

each page is placed in the tree based on the programmers' navigation history, it is possible for two different branches to contain the same page if the programmer navigated to that same page via different seed pages.

Defining Branch Visits: A branch visit contains an uninterrupted sequence of page events that occur within a single information branch (see Figure 3). To identify branch visits from our recorded log data, we first extract all of the pages visited in the log files and their access times. We note that because the running program also appears as a webpage via localhost, the page visit information includes both information seeking activity and testing activity. Then, we define a new branch visit. We find the branch for the first webpage inserted. Afterwards, as long as subsequent webpages are from the same branch, we add them to the branch visit. Once a webpage is from a different branch, it ends the current branch visit and begins a new branch visit (see Figure 3). Testing activity, the creation of a new branch, or revisiting a page from a previously created branch will all generate a new branch visit.

6.1. Qualitative Coding Support

To assist in the labeling process, we created: 1) a branch visit visualization that enabled coders to view the webpages visited and code changes related in time, and 2) a spreadsheet that aligned with the branch visit visualization to enable labeling of branch visits.

6.1.1. Branch Visit Visualization

The branch visit visualization is a webpage that displays a timeline of branch visits, as well as the state of the web browser and code editor at the selected time. Figure 4 shows the components of the branch visit interface. The interface enables researchers to view each of the webpages visited by a participant, click links to see the live versions of webpages visited by participants, and see which branch the webpages are associated with. Alongside the webpages, the interface shows the participants's code at each point in time, enabling the researcher to see how the information context relates to participants' code changes.

6.1.2. Coding Spreadsheet

To reduce the workload associated with our coding process, we automatically generated a coding sheet containing an overview of a given programmer's API learning process by iterating through each branch visit in

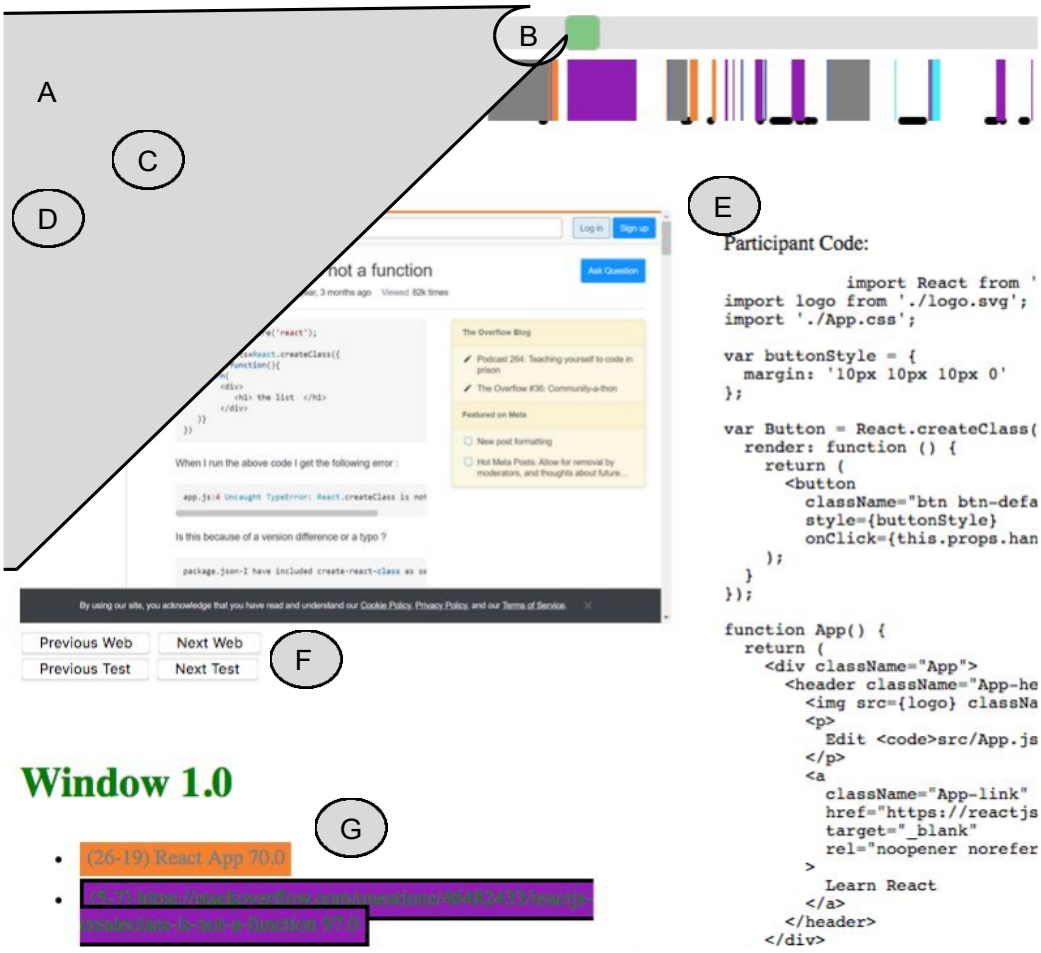

Figure 4: We used this branch visit visualization to view the webpages viewed within branches and corresponding participant code. (A) Branch visit time blocks. (B) Moveable slider that selects which branch visit to view. (C) Time of current branch visit. (D) Current webpage based on the selected timepoint. (E) Participant's current code based on selected timepoint. (F) Buttons that move the slider to the next webpage visited or next code test. (G) Currently open webpages with links and branch numbers. Their background colors match the branch view time block colors in (A).

order. For each branch visit, we recorded the starting time, branch id, and branch seed (which is either a URL or a set of search terms). This enables the researcher to focus purely on the sensemaking stages through the coding process.

It is important to note that not all branch visits actually correspond with

foraging behavior. Prior research suggests that some of the page visits that occur during API learning consist of programmers quickly flipping between multiple pages without reviewing information on those pages [14]. This behavior contrasts with longer page visits in which programmers perform new foraging activities. To differentiate these two, we also recorded a visit type. We classified branch visits in which the average page visit length was less than 2.5 seconds (less than the amount of time necessary to read a sentence) as flips. Branch visits in which the page visits were longer than 2.5 seconds, we classified as forages. We excluded flips from this analysis unless they led to a code change, suggesting that a user visited them for a particular reason.

7. Labeling Branch Visits with Sensemaking Activity Labels

To connect branch visits with stages in sensemaking, we developed a coding system inspired by Klein et al's sensemaking model [57, 59]. After developing an initial coding system, two authors independently labeled all branch visits and code tests from three pilot users, discussing and iterating on labels and descriptions until there was consensus on the labels and definitions of the labels. Because each data point could have multiple labels, we used Krippendorff's alpha with Jaccard's distance to measure inter-rater agreement. After the two researchers reached an acceptable inter-rater agreement on the pilot data, $\alpha = .801$ [60], the two authors labeled the twelve participants' branch visits independently.

We briefly describe each of Klein's sensemaking stages and detail the labels we used to capture that stage [8]. The labels describe activities that occur in the information context, the code context, and across both, as shown in Figure 5. Our high-level labels correspond to six sensemaking stages: map, seek, elaborate, preserve, reframe, and test. Each label has several variations that capture the results of the sensemaker's actions in that stage, resulting in multiple possible sub-values for each label. In this section, we introduce the six stages and their variations.

7.1. Activity Labels

Mapping Data to Frame (Map): During this stage in the sensemaking model, the sensemaker creates a relationship between the data and a frame that helps to organize and explain that data. In the API learning setting, the data consists of resources found on the web. The frame consists of a plan to achieve the learner's current goal. The existence of code modifications

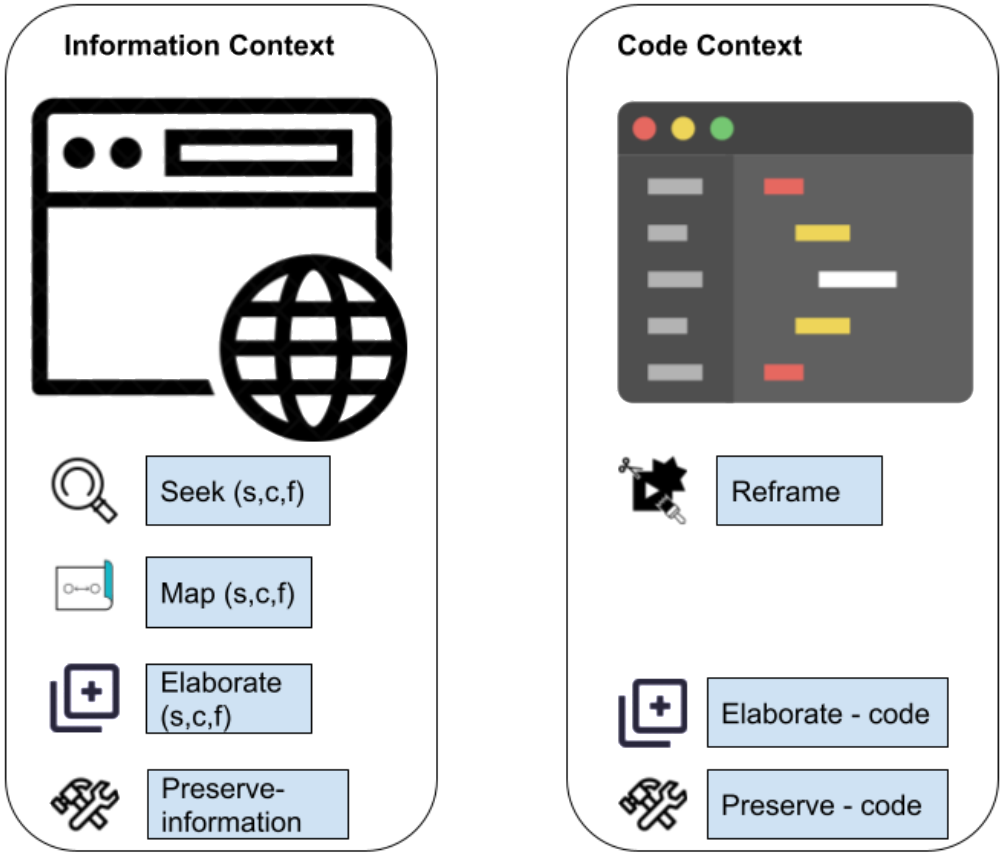

Figure 5: Labels for qualitative coding based on sensemaking theory and their contexts. Seek, and map exclusively occurred in the information context. Reframing occurred exclusively in the code context. Both elaborating and preserving incorporated activities that could take place in both contexts: information search activities occurred in the information context and the code changes related to elaborating and preserving occurred in the code context.

serves as a signal that a frame exists. However, programmers' understanding of their code may vary substantially. While some programmers may have a detailed understanding of each element, others may have a general belief such as "This code will connect to a database and send a query." Additionally, programmers' beliefs about their code may be incorrect. Regardless of the correctness of the resulting frame, the mapping process results in a code modification that programmers hypothesize will cause or contribute to a goal behavior.

Precondition: The programmer begins the mapping stage without an existing causal relationship and expresses a specific information need, typically through search terms.

The mapping stage has three possible end states:

- Success (s): The programmer identified a frame and modified their starting code.
- Continuance (c): The programmer has added a new term to the subsequent search that adds detail to their search process, but has not made any code modifications.
- Failure (f): The programmer did not identify any usable information.

Seeking Frame (seek): Sometimes sensemakers may not be able to identify an initial information goal, preventing them from beginning with the mapping stage. In the programming context, programmers may instead seek general information that is not related to a particular subtask. For example, programmers might navigate to the React documentation or search for a tutorial or introduction. By exploring overview material, programmers may then be able to articulate an initial information goal and progress to the mapping stage.

Precondition: The programmer goes into this phase without an existing frame. If they perform a search as part of this phase, the search contains no subgoal information related to the task.

The seeking stage has three possible end states:

- Success (s): The programmer identifies a task-related information goal. Typically, successful seeks are followed by mapping.
- Continuance (c): The programmer identifies a new term that adds detail to their search process, but does not link to the task.

- Failure (f): The programmer does not identify any usable information. Their next task begins without a defined frame and does not introduce new terminology.

Elaborating the Frame (elaborate): A starting frame typically explains some of the observed data, but not all. When elaborating, the sensemaker expands an existing frame with additional details. None of the details and theories in their current frame are replaced, but more supporting information is added. In the context of programming, elaboration will typically take the form of a new search with new terminology that relates to exist-ing knowledge in the frame. For example, a programmer who already has a textbox in their code might search for React keywords input and setState to try to figure out how to capture information from a textbox and save it in a state.

Precondition: The programmer goes into this phase with at least one existing causal relationship; there is a frame. Their search contains new API terms with a relationship to the existing frame.

The elaborating stage has four possible end states:

- Success (s): The programmer emerges with a new mapped element that can be added to their frame.
- Continuance (c): The programmer emerges with another new but related term that adds detail to their search process, but does not have a frame. The related term appears in a subsequent mapping action.
- Failure (f): The programmer doesn't emerge with anything that they can use. Then, their next search takes a different approach.
- Code: the programmer takes content from the successful elaboration and modifies their code with it. A code elaboration typically occurs alongside a successful elaboration, but can also occur independently, as some programmers may have existing knowledge that they may connect to their frame.

Preserving the Frame (preserve): In some cases, sensemakers will notice a mismatch between the data and their current frame. When sensemakers conclude that the data is the source of these inconsistencies and explain away data that does not match the frame, this is called preservation. This can be a legitimate course of action in response to transient or faulty

data. At a core level, the sensemaker is choosing to maintain their current hypothesis. In the programming context, preservation processes begin when the programmer receives data that suggests that their frame is incorrect or incomplete in some way. This typically occurs when a programmer has attempted to integrate new code into their project based on found information and has encountered a problem. The code generates an error or it doesn't work as expected. This sets off an attempt to fix the problem, either by editing the code directly or by initiating a round of information foraging aimed at troubleshooting the problem, preserving the core solution and the frame that the programmer has established.

The preserving stage has two possible end states:

- Info: The programmer seeks information to try to resolve the issue with their frame. The programmer may begin a new search for information related to their problem or return to the information space where they originally found their frame. This label can refer to successful or unsuccessful information searches.

- Code: The programmer emerges with a modified frame with a potential code change that may address the error. Code modification can occur alongside an info action, in which case the preservation likely results from information found, or independently. Independent changes are typically based on prior related knowledge.

Reframe: Reframes occur when the programmer determines that their previous frame is no longer valid and replaces it. The Klein model includes three stages that explore different aspects of considering multiple frames: questioning the frame, comparing frames, and re-framing. Broadly, the three capture the sensemaker's different levels of certainty regarding the relative merits of the frames being compared. Given behavioral data, we cannot determine the sensemaker's level of confidence in each of frames considered, so we have grouped these three into a single Reframe stage. In the Reframe stage, the programmer replaces their existing frame with a new one. In some cases, a participant puts one frame on hold and while searching for a new potential frame. Once the programmer finds a new potential frame, they replace the held frame in their code. While we do not have an explicit label for this, our reframe labels and their context enable us to determine when participants put one frame on hold while researching another, behaviors that correspond to Klein's stages of questioning the frame and comparing frames.

The re-framing stage has four possible end states:

- Delete (d): The programmer deletes or comments all of their code, effectively removing any frame they may have been working on. Because the frame serves as a skeleton on which the solution is built, deleting it effectively requires starting fresh.
- Original (o): The programmer returns their code state to the original state of the starting code template for the task, removing any frame they may have been working on.
- New (n): The programmer introduces a new frame that has not previously been in the code.
- Previous (p): The programmer returns their code to a previous state or re-introduces a frame that they previously worked with.

Code Test (test): While testing is not an explicit part of prior sensemaking models, the act of testing code is an important way that programmers obtain feedback. The feedback they receive is often what triggers the shift to a new activity, so we felt that it was important to include explicitly. It is arguably most closely related to questioning the frame, in which the sensemaker notices inconsistencies between the data and their current frame, suggesting an error. In cases where testing reveals an error, the programmer may not know the source of the error, but the inconsistency serves as a cue to investigate further.

The code testing stage has four possible end states:

- NA: The participant tests the template code without any changes or other empty code.
- Failure (f): The tested code is immediately removed.
- Continuance (c): The tested code has problems that the programmer next tries to address.
- Success (s): The tested code appears to work and the programmer moves next to a new subgoal.

8. Defining Sensemaking Episodes

Through our labeling process, we labeled each activity (foraging within a branch, coding, and testing) programmers performed with a sensemaking stage. However, these individual activities are often seconds in length and represent steps within a larger process. Often, when there are multiple of the same label in a row, the user is working toward one subgoal with multiple similar actions. To understand the overall sensemaking process, we grouped individual sensemaking activities into larger sensemaking episodes that attempt to capture the activity related to a subgoal. These episodes enable us to explore the high-level process and flow through sensemaking phases. Generally, sensemaking episodes are made up of sequential activities of the same type, like multiple seeks in a row or multiple maps in a row. However, they can also include activities from other sensemaking stages, or embedded episodes, if they occur before the subgoal concludes. We define the sensemaking episodes as follows:

8.1. Seeking Episode

A seek episode begins with the first time that a programmer performs a seek of any type and ends successfully when the programmer identifies a subgoal that can be used to construct a frame. This is demonstrated with a map occurring on the same page and a related code modification. Seek episodes can also end with a transition to another activity such as mapping, preserving, or elaborating.

8.2. Mapping Episode

A map episode also begins with a map activity of any type. Most of-ten, this is a search that corresponds to a task subgoal such as creating a React button. The mapping process ends when the programmer has suc-cessfully created a map (reified through a new frame) consisting of a code based hypothesis. For example, a programmer might add a code snippet that he or she believes will create a new button in the current application. This hypothesis may or may not be correct. Mapping episodes can also end with and include the programmers' transitions to another activity: seeking, preserving, or elaborating.

8.3. Elaborating Episode

An elaborating episode begins with any type of elaboration activity in which the participant searches for information or adds code related to a new subgoal. A successful elaboration process ends when the programmer has modified their code with new functionality. Elaboration episodes can also end with (and include) a transition to seeking or mapping.

8.4. Preserving Episode

A preservation episode begins when a programmer tests a program that does not yet work, but does not remove the not-yet-functioning code. Preservation ends when a programmer tests a working version of that code.

9. Results

Our results suggest that there are patterns in how programmers move through the sensemaking process. These patterns both point to where programmers need the most help and have implications for how to support programmers learning an API. Figure 6 shows a summary of how programmers transitioned through different types of episodes as they worked on their tasks. We discuss our results in terms of sensemaking episodes, or groupings of the individual sensemaking labels, that let us talk about the process at a high-level.

9.1. Summary of programmer sensemaking processes during API learning

All programmers' early tasks included map episodes: goal directed searches for information that, when successful, resulted in a hypothesis about code that would accomplish some piece of their target task. 8 of 12 programmers also incorporated seek episodes early in the task process, before they had added code. These were non-task directed searches for information, but 5 of the 17 seek episodes led to programmers finding a map. 15 of the 17 seek episodes occurred when programmers' code was in the starting state (12 episodes before adding code, 3 after removing unsuccessful code). We present an aggregate view of the the transitions between sensemaking states that our programmers made. We selected this aggregate view in order to highlight the most common kinds of transitions that occurred through the completion of the React task.

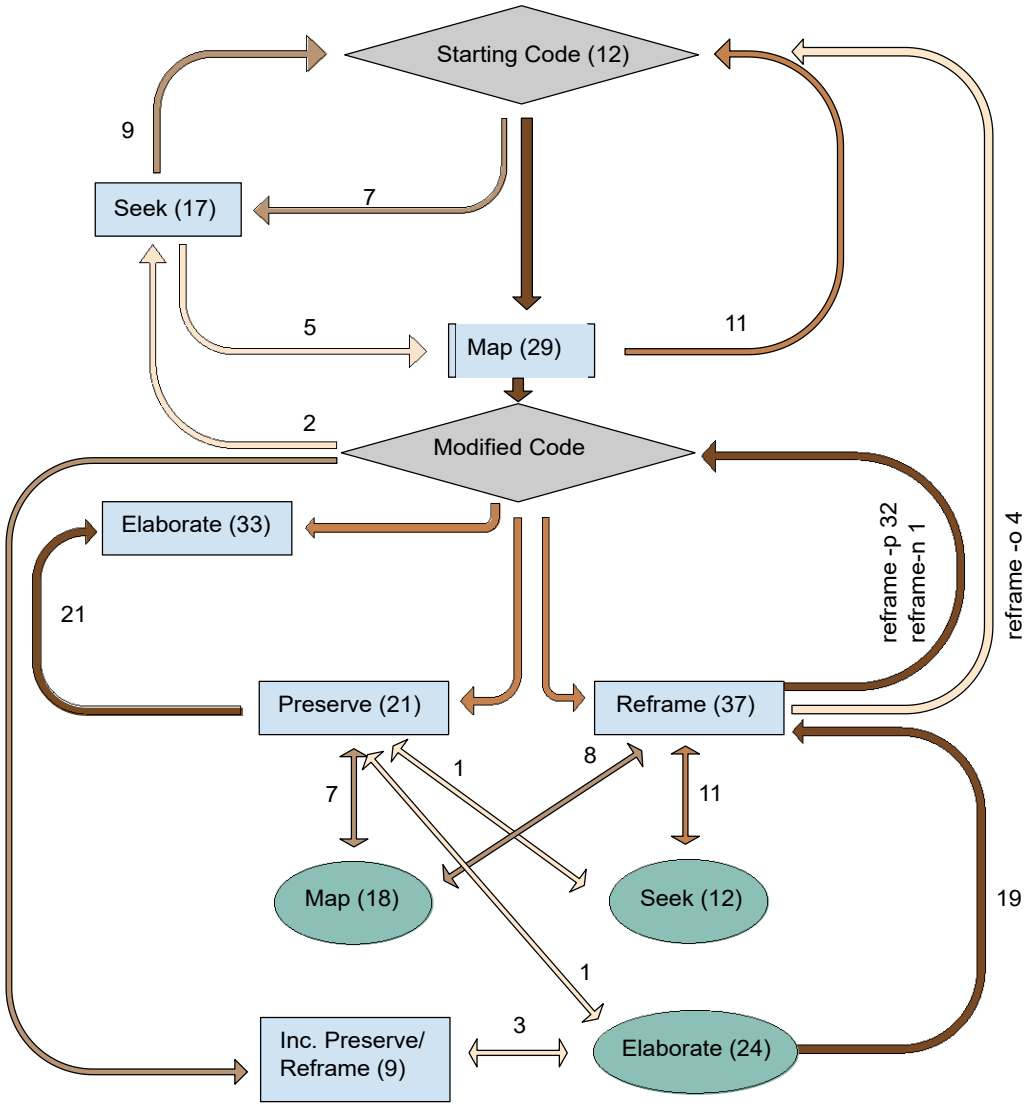

Figure 6: Aggregate overview of programmers' programming process through sensemaking episode types. Darker arrows indicate more common transitions. Episodes are shown as blue rectangles with a frequency count. Embedded episodes, episodes that occur within another episode type, are shown as green ovals.

When leaving map episodes, programmers progressed primarily to one of three episode types: preserve, reframe, and elaborate. Programmers' behavior after the initial stages largely centers on fixing broken code. Both preserve and reframe episodes began with broken code that programmers attempted to fix successfully and unsuccessfully, respectively. However, even 13 of the 33 elaborate episodes began with non-functional code (39%). Thus the issues of non-functional code arose even as programmers attempted to add new functionality. Programmers' attempts at debugging this broken code occurred during preserve and reframe. Behavior during debugging was fairly complex and included embedded episodes (shown in Figure 6 as ovals rather than rectangles) corresponding to map, seek, and elaborate behavior. Embedded episodes are complete episodes of another type that occur within an enclosing episode. Embedded seek episodes occurred as programmers attempted to understand API elements in broken code. Embedded map and elaborate episodes occurred as programmers replaced their existing solutions with new ones or augmented their code with new capabilities while debugging their current solution. Interestingly, while preserve and reframe episodes were about equally likely to include embedded map episodes, reframe episodes were much more likely to include seek and elaborate episodes, suggesting that we may be able to identify debugging behavior that is at risk of not succeeding.

Below we discuss the behaviors associated with each episode type in greater detail.

9.2. Mapping

Participants completed an average of 2.4 independent map episodes (SD=1.8). Independent map episodes averaged 2.1 activities (SD=1.5) Additionally, participants completed an average of 1.6 (SD=1.2) map episodes embedded in preserve and reframe episodes. Embedded episodes averaged 1.3 activities (SD=0.9). Maps occurred in two places: 1) at the beginning of the task and 2) as programmers attempted to debug broken code.

Longer mapping attempts were often unsuccessful. Of the 29 map episodes, 19 resulted in successful maps. 9 were the result of new, task-based information, and 5 included returns to existing task-based information. Failed map episodes tended to require more steps, averaging 3 as compared to 1.7 for successful maps, and consulted multiple types of resources in a single episode. For example, a multiple resource episode might include a new task-based search and a return to a previously found resource. 8 of the 10

failed mapping attempts included a mix of resource visits in a single episode. Among the successful maps, only 1 of the 19 included multiple resources.

Programmers struggled to build more than half of the initial maps they made. While participants were successful in creating maps, the existence of more than 2 successful independent mapping episodes per participant means that, on average, participants discarded a successful map and started again from scratch before they were able to build on their map. While a map represents a programmers' hypothesis that a given section of code will accomplish a target goal, that hypothesis is not always correct. Programmers frequently left map episodes with unworkable code due to combining code with differing syntax, version incompatibilities, or use of non-React API elements. When programmers failed to get their code working (through a subsequent reframe episode), they reframed back to a previous code state.

Embedded maps occurred as part of debugging. Programmers completed 18 additional map episodes that were embedded in preserve and reframe episodes. In these episodes, mapping occurred when a programmer found an alternative code snippet to accomplish a goal and replaced their current code. This action suggests that programmers were considering multiple frames simultaneously and replacing an unsuccessful frame with an alternative approach. Embedded maps were split nearly equally between preserve (7 episodes) and reframe (8 episodes), with an additional 2 incomplete episodes.

9.3. Seek

Participants completed an average of 1.42 independent seek episodes (SD=1.38). Independent seek episodes averaged 3.8 sensemaking activities (SD=2.8). Additionally, participants completed an average of 1 (SD=1.35) seek episode embedded in preserve and reframe episodes. Embedded episodes averaged 2.3 sensemaking activities (SD=0.8). A majority of seeks (all but 2) occur in one of two distinct contexts: 1) early in the task process as programmers build their initial maps and 2) later in the task process as users struggle to debug broken code.

(1) Programmers used seek when starting. Overall, we found 71% of seeks occurred when code was unchanged (i.e. programmers had not yet added any functionality to the starting template). Of these, 12 occurred before programmers had modified the starting code and 3 occurred after programmers had reverted back to the starting code. 4 of the 15 were successful in leading participants to a mapping attempt. Further, our results suggest that above average use of seek at the beginning may indicate struggle.

Two participants who ultimately made little progress towards the target task started with and repeatedly attempted seeks. Together, they are responsible for a combined 8 of the 15 getting started seeks.

Programmers mixed gathering task-focused and overview information. Programmers sought general information before task-specific information in most cases. 7 seek episodes of 17 total episodes contain only foraging in new branches. Thus, all resources were the results of seeking overview information. An additional 3 seek episodes mix foraging in new branches with foraging in the results of previous seek episodes. However, seek was not always programmers' first step. 7 seek episodes include vis-its to existing task-focused branches. This suggests that programmers first searched for task-focused information and then attempted a seek afterwards, perhaps to gain insight into API elements and concepts encountered through task-focused searches. The approach of mixing task and non-task focused searches at the beginning was less successful: 0 of the 7 seek episodes that included returns to task-focuses searches were successful. 4 of the 10 that focused on overview information ultimately led to a map episode.

Programmers attempted embedded seek episodes when struggling to repair broken code. While nearly all of the independent seek episodes occurred at the beginning of programmers' tasks, we saw 18 seek episodes that were embedded within preserve and reframe episodes. In these episode types, programmers attempted to repair non-functional code. When they could not repair their current code, programmers searched for new frames to replace their existing ones. Interestingly, the use of seek was substantially more frequent in reframe episodes than in preserve episodes, occurring 11 times and 1 time in reframe and preserve, respectively. 5 of the 12 embedded seeks were successful in leading to an embedded map episode. Participants incorporated return visits to information from prior searches in all but one of these episodes. However, it is notable that episodes with return visits to non-task focused information were more likely to be successful (4 successful episodes of 6 with returns to non-task focused branches vs. 1 successful episode of 5 with returns to task-focused branches). This may indicate that conceptual information may be more valuable than task-focused information during failing repair attempts. Broadly, in our data, the existence of embedded seek episodes predicted that programmers would ultimately fail to repair their current non-functional code.

Programmers rarely performed seek episodes after mapping. Only 2 of the 17 seek episodes occurred after programmers had moved past

the starting code state. Neither was successful in leading to a mapping attempt. In both cases, participants had working code. Both episodes include visits to previously opened task-focused branches, suggesting that, like the 44% of the getting started tasks, programmers used seek to research API elements and concepts.

9.4. Elaborating

Once programmers have an initial mapping that solves a piece of the larger task, they attempt to extend their initial program with additional sub-tasks through elaboration. Elaborations were the most common type of episodes overall, with a total of 33 independent episodes (M = 2.8, SD = 2.1) and 24 episodes embedded in preserve and reframe episodes (M=2, SD=1.3). Independent elaborations averaged 3.5 activities (SD=3.5). Embedded elaborations averaged 1.5 activities (SD=1.2). 19 of the 24 embedded elaborations occurred in a single activity, a return visit to an existing web resource. All completed episodes ended with a successful elaboration activity.

Programmers frequently began elaborations with broken code. 39% of independent elaborations and 83% of embedded elaborations began with broken code. In both cases, this suggests that programmers are seeking to incorporate new functionality on top of non-functional code. It was unclear from the behavioral data why participants tried to integrate new functionality into a broken codebase. However, in the case of embedded elaborations, this may represent a form of last ditch effort to repair broken code, as elaborates occurred almost exclusively in reframe episodes.

Elaborations leveraged task-focused resources. Independent and embedded elaborations both heavily leveraged task-focused resources. 21 of independent elaborations referenced new task-based resources and 18 referenced existing task-based resources as compared to 4 existing non-task based resources and 0 new ones. Embedded elaborations referred to previously found task-focused resources (19 episodes), with references to newly found task-focused resources appearing in 5 episodes. Non-task focused resource use was rare: 2 episodes referenced previously found non-task focused resources and 1 referenced a newly found resource.

Mapping example incompatibilities lead to errors. For both single-activity elaborations and multi-activity elaborations, the code changes made were frequently incorrect or unusable in the participant's code. The errors reflect the kinds of code incompatibilities found in the mapping episodes: differing syntax style, outdated examples, and use of non-React API elements.

However, we also observed an additional issue: the addition of incomplete code snippets. For instance, a snippet might show the use of a state variable to hold information but not include the initialization of that state variable. These incompatible and incomplete code additions led to subsequent preserve or reframe episodes.

9.5. Reframing

Reframe episodes represent cases where the programmer attempted to repair non-working code and failed. These episodes ended with the programmer removing some or all of the non-working functionality. Reframes are particularly interesting from an API usability standpoint because they represent situations where a programmer struggled to achieve a goal and ultimately gave up. Reframing episodes occurred 37 times, making them the second most common episode type behind elaborating. Participants' most commonly ended reframing episodes by reframing to a previous frame (reframe-p 32 times). Programmers also reframed to the original frame (reframe-o) 4 times and to a new frame 1 time. Reframing episodes were expensive, requiring an average of 7.5 activities (SD=7.8) to complete. Given that reframing is both common and time intensive, this represents a high cost to the programmer, as much of the reframing process is, by its nature, a failure.

Programmers revert to previous code states without attempting to repair. In 43% of reframing episodes, programmers do not attempt to repair the code. They simply replace it (6 episodes) or test it and then replace it (10 episodes). This further emphasizes the high cost of reframing when programmers do attempt to fix their code (M=12.0 activities, SD=7.6).

Reframe episodes are likely to include embedded episodes, especially seek and elaborate. The high cost of reframe episodes is in part due to embedded episodes of other types: map, seek, and elaborate. Reframe episodes include a unique mix of embedded episodes. 11 of the 12 embedded seek episodes and 19 of the 24 embedded elaborate episodes occurred during reframes. In other words, when programmers struggled to fix broken code, they were more likely to 1) seek general information and 2) attempt to extend the functionality of their programs. Because of their uneven distribution, the use of embedded seeks and elaborates suggested that a given episode would ultimately fail. However, this behavior also suggests that reframing is a point during which programmers are open to reading material about the concepts and best practices associated with a given API, assuming that they can find the right avenue to pursue.

Programmers often reframed back to previous non-working states. It is unclear why programmers elected to reframe to non-working states. It is possible that, due to the unfamiliar context, programmers do not accurately remember whether a particular previous code state was functional. Alternatively, this behavior may reflect a foraging-like behavior in the code space. Programmers may explore one strategy to solve a problem, remove part of the solution, and try a different approach to complete it. Regardless of the motivation for reframing back to non-working code, it is important to note that reframing back to a previous state can include partial progress towards a sub-task.

Many copied code snippets contain multiple errors. We noted that many of the code snippets participants copied into their code contained multiple issues. For example, a code snippet might be both using an incompatible or outdated syntax style and referencing a state that does not exist. In order to see progress, some code issues should be addressed before others. However, we observed cases in which a programmer selected an issue to work on that was masked by another existing problem. Thus, even when they made changes that addressed an existing problem in their code, those changes did not always immediately change the output of the program. This seemed to lead some participants to conclude that their actions were not helping and reframe their code back to a previous state.

9.6. Preserving

Successful code repair attempts (preservation episodes) ended with a tests in which the new functionality worked. Participants' programming sessions included a total of 21 preservation episodes. Each episode required an average of 8.6 (SD=7.4) steps to resolve, making it an expensive process. Additionally, there were 36 instances in which programmers successfully tested new code without needing to make any modifications.

Programmers preferred to resolve code issues with information rather than exploration. 18 of the 21 preserve episodes relied on information to resolve issues with code. 14 returned to existing task-based resources, 9 consulted new task-based resources, and 7 searched for information on error messages. Only 3 episodes were resolved by testing and editing code alone.

Successful resolution of code issues includes few embedded seek and reframe episodes. Where reframe episodes included a significant number of embedded seek and elaborate episodes, these were largely absent when programmers succeeded in repairing broken code. Preserve episodes included

only 1 of the 12 embedded seek episodes and 1 of the 24 elaborate episodes. Preserve episodes did include 7 embedded map episodes during which programmers found alternative code to accomplish a goal and replaced their working code.

10. Discussion and Future Work

Our sensemaking analysis of API learning provides new insight into the kinds of information programmers search for and how they shift over the course of attempting to build a simple React program. We discuss reframing and preservation as the stages of the sensemaking process most in need of better support and highlight the dificulties around repairing code. We then summarize implications for documentation, tools, and education. Finally, we briefly discuss the potential generalizability of our coding scheme.

10.1. Information Needs Vary by Sensemaking Stage

Programmers were most open to API overview information at two points: 1) when they were first starting, before they had begun to write their own code and 2) when they were struggling (and failing) to get a version of their code to work. During the initial period, engagement with overview materials tended to be short. Programmers appeared to want a sense of React, but largely skimmed the materials they found. When they returned to overview material during a debugging process, it seemingly served as a last attempt to find a path forward.

During other sensemaking episodes, programmers tended to search for and attempt to use task-based resources. Their approaches were greedy - programmers appeared to use the first promising-looking resource they found that addressed their current subgoal. The limited number of activities in mapping and elaboration suggest these initial uses of found code came with a shallow understanding of that code. This is further supported by the fact that both initial and elaborated code were often non-functional and led to lengthy preserve and reframe episodes. During these episodes, programmers revisited task-based resources and, in the case of reframes, sought additional overview information.

Our sensemaking analysis identifies how and when programmers seek out particular kinds of information as well as the types of tasks on which they struggle.

10.2. Reduce Reliance on User Question Generation During Debugging

Prior work in API learning has suggested that programmers can struggle to frame a question that leads them to relevant information [12, 13]. In our study, the dificulties around framing a question coincided with debugging broken code. Programmers tended to be successful in searching for and finding information about the basic elements of the API such as user interface elements and events. However, in both preserve and reframe episodes, programmers typically had erroneous code due to combining incompatible code or the absence of needed setup code. In both cases, framing a helpful question was dificult. In the most successful cases, programmers compared their code with found example code to reason about potential issues, but we note that programmers failed to repair their broken code nearly twice as often as they fixed it.

Tools that enable programmers to ask syntactic and structural "What's wrong with this?" questions based on faulty code, like the Whyline [61], could be especially important in early API learning. Given the availability and use of question-answer sites, enabling automatic creation of bottom-up resources based on code seems an important challenge for future work.

10.3. Support Needs for Reframe and Preserve

The reframe and preserve episodes were the most expensive of all of the episodes in terms of programmer activity. Improving the eficiency of both episode types and increasing the proportion of preserves could dramatically improve the amount of time it takes programmers to complete a task in an unfamiliar API. Further, reframing may result in high extraneous cogni-tive load, limiting the potential for programmers to learn during this phase. Programmers entered into the reframing process after adding new code via mapping new frames and elaborations. But, in the reframing process, the majority of these new code additions were removed without ever reaching a working state. As a consequence, programmers never figured out what would have been necessary to get them working and never understood why their proposed solutions did not work as expected. These proposed solutions are particularly important because they represent an instantiation of a programmer's model of how the API is supposed to work. When they are deleted before reaching a working state, the programmer does not have an opportunity to revise their underlying model appropriately. Taken together, this represented a perfect storm of extraneous load. Programmers invested significant time in solutions that do not contribute towards their task. This

time investment likely contributed very little to their API understanding as the kinds of feedback necessary to achieve a working state were not available. Future research that explores both how to reduce the amount of time programmers spent reframing and how to provide better support for learning when reframes do happen is an important design challenge.

10.4. Programmers Lack Strategies for Repairing Broken Code

Programmers in our study quickly sought example code that contributed to an initial goal, and spent little time reading content presented alongside their found example code. Frequently, this copied code did not work. Our results suggest that programmers may not have an effective set of strategies for repairing broken example code. Programmers primarily responded to broken code in one of three ways: by immediately removing it, by viewing general API information (seeking), and by adding new lines to the program (elaboration). In all three cases, we believe these actions indicate uncertainly around how to proceed. Immediately removing the code serves as a tacit acknowledgement that a repair attempt is likely to fail. If failure is the most likely outcome, the best strategy is to immediately move on to a new example. Based on our study data, the assumption of failure is a fairly accurate assessment. Seeking general overview information often appeared to be motivated by a hope that the programmer would encounter information that might spark a solution. Unfortunately, this rarely happened. In fact, the presence of an embedded seek activity was a strong signal that the repair attempt would ultimately fail. Finally, some programmers added new code to their existing, broken, solution. When doing this, they were most likely to return to a previously found webpage and the source of the example. In this case, we suspect that programmers may believe that they have missed something within the example, rendering the example incomplete. This approach was also rarely successful.

However, it is also important to note that much of the copied code was close to working. Code examples with missing code elements (such as declaration or initialization statements) and ones that were out of date were common. These examples were likely the most problematic for new API users. More experienced API users would likely have been able to identify and correct the issues. But our novice API users appeared to want to invest mental effort in an example only after they had determined that it did something useful.

10.5. Implications for Documentation

Prior research finds that programmers often struggle to extract needed information from documentation [4, 5, 12]. In our study, programmers primarily appeared to seek information in two categories: 1) task-related examples and 2) API overview information. The contexts and motivations for this accesses have some implications for the design of future API documentation.

Programmers in our study were largely successful at finding relevant task-related examples, typically via search, but struggled to repair broken examples. Documentation that provides examples first but provides pathways for further exploration may be helpful. For example, providing connections from examples to relevant API concepts may help programmers to reason about potential issues in their found code. Additionally, we noted that some of the problems that programmers encountered were due to outdated examples. Example troubleshooting materials that aim to help programmers identify and update deprecated examples may help to address this.

Programmers predominately sought overview information at points when they were not sure how to take a task-related step. This occurred both at the beginning of the process, before they had written any task-related code and, later, when they encountered non-working code that they did not know how to repair. Thus, materials that define and describe the important concepts in an API and link them to code examples may help to guide programmers towards a concrete and productive step to take.

10.6. Implications for Tools

Tool support that helps programmers to repair non-working example code is an important area for future research. When faced with a non-working example, it is dificult to evaluate what issues may be causing the code to fail, particularly without knowledge of the API. In our study, this struggle manifested itself through the high cost and low success rates of reframe and preserve episodes. Future work should explicitly address the challenge of broken code repair, which has been under-explored. We believe that tools that leverage information about the necessary structure for API-related programs, the history of API changes, and the large collections of shared code may prove valuable in suggesting corrections for non-working code.

10.7. Implications for Education

Much of education research has focused on formal education contexts in which an instructor guides students through a gradual introduction of

concepts and related code. While learning in formal contexts is clearly important, we also need to understand and effectively support learning that occurs outside of this context. Research suggests that programmers prefer just-in-time learning approaches [3]. Our results suggest that programmers may benefit from learning strategies around code repair, which may include how to identify relevant API concepts from a broken code example. However, it is important to note that without studies that fully document the causes of broken code across a range of different APIs, it is not clear that effective, generalizable strategies for code repair are known.

11. Generalizability of Coding Process

While our coding scheme was developed within the context of novices learning the React API, we believe that it will capture processes of just-in-time learning that occur in the context of constructing code across different APIs and different programmers with differing experience levels. We would expect to see differences in the dominant sensemaking stages and the lengths of individual stages between different APIs and programmer experiences levels. However, the coding process is labor intensive. Using the branch visualization helped by providing a single context with the web resources being viewed, the search they originated from, and the changes made to the programmers' code. Still, we estimate that we spent about 2-3 hours per hour of participant time in appropriately labeling code. This is in part due to the need to understand what the code is doing and whether or not the change was successful in order to appropriately classify it. Some additional automation of this process is possible, which may streamline the labor necessary to test successive code changes made by programmers.

12. Limitations

There are a number of potential threats to the validity of our results. The results reported in this paper represent a narrow sliver of the overall programmer population: predominantly male, twelve later stage undergraduate students completing a single task with the React API. It is unclear to what degree their behavior reflects a broader population of programmers and range of APIs and tasks. Further, the experimental setup may have affected their behavior. While other research suggests that the task-oriented focus is common among programmers [3] and that this task-focus is not unique

to programming [62], it is possible that our participants may have behaved differently in an authentic situation such as needing to learn React for a job. While our results are suggestive, it will be important to perform studies with a broader set of programmer experience levels, a diverse audience of programmers, and a variety of different APIs to ensure that the findings here generalize.

13. Conclusion

This paper uses the lens of sensemaking to explore how a group of twelve student programmers interacted with information at different stages completing a task with an unfamiliar API. Our participant programmers first attempted to find and use task-focused information with shallow understanding. Attempts to more deeply understand code arose when debugging broken code, suggesting a need for resources that enable understanding to flow outwards from code. While programmers did search for overview information, they did so briefly when first starting and later when failing to debug broken code. Further, our methodology enables us to quantify some of the struggles that programmers encounter. This allows us to identify and prioritize the challenges that have the greatest potential for impact.

14. Acknowledgements

This paper is based in part upon work supported by the National Science Foundation under Grant IIS-2128128."

References

[1] Programmable Web: API directory (2018).
URL https://www.programmableweb.com/category/all/apis

[2] W. Santos, Research shows interest in providing APIs still high — ProgrammableWeb (2018).
URL https://www.programmableweb.com/news/research-shows-interest-providing-api

[3] J. Brandt, P. J. Guo, J. Lewenstein, S. R. Klemmer, Opportunistic programming: How rapid ideation and prototyping occur in practice, in: Proceedings of the 4th international workshop on End-user software engineering, ACM, 2008, pp. 1–5.

[4] M. P. Robillard, R. Deline, A field study of API learning obstacles, Empirical Software Engineering 16 (6) (2011) 703–732.

[5] M. P. Robillard, What makes APIs hard to learn? Answers from developers, IEEE software 26 (6) (2009) 27–34.

[6] J. Stylos, B. A. Myers, Mica: A web-search tool for finding API components and examples, in: Visual Languages and Human-Centric Computing (VL/HCC'06), IEEE, 2006, pp. 195–202.

[7] S. Oney, J. Brandt, Codelets: Linking interactive documentation and example code in the editor, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 2697–2706. doi:10.1145/2207676.2208664.
URL https://doi.org/10.1145/2207676.2208664

[8] G. Klein, J. K. Phillips, E. L. Rall, D. A. Peluso, A data–frame theory of sensemaking, in: Expertise out of context, Psychology Press, 2007, pp. 118–160.

[9] B. Dervin, Sense-making theory and practice: an overview of user interests in knowledge seeking and use, Journal of knowledge management 2 (2) (1998) 36–46.

[10] D. M. Russell, M. J. Stefik, P. Pirolli, S. K. Card, The cost structure of sensemaking, in: Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '93, ACM Press, New York, New York, USA, 1993, pp. 269–276. doi:10.1145/169059.169209.
URL http://portal.acm.org/citation.cfm?doid=169059.169209

[11] P. Pirolli, S. Card, Information foraging in information access environments, in: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press/Addison-Wesley Publishing Co., 1995, pp. 51–58.

[12] E. Duala-Ekoko, M. P. Robillard, Asking and answering questions about unfamiliar APIs: An exploratory study, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 266–276.

[13] C. R. Rupakheti, D. Hou, Satisfying Programmers' Information Needs in API-Based Programming, in: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, IEEE, 2011, pp. 250–253.

[14] G. Gao, F. Voichick, M. Ichinco, C. Kelleher, Exploring programmers' api learning processes: Collecting web resources as external memory, in: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020, pp. 1–10.

[15] C. Kelleher, M. Ichinco, Towards a model of API learning, in: 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2019, pp. 163–168.

[16] A. Horvath, S. Grover, S. Dong, E. Zhou, F. Voichick, M. B. Kery, S. Shinju, D. Nam, M. Nagy, B. Myers, The long tail: Understanding the discoverability of api functionality, in: 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2019, pp. 157–161.

[17] M. J. Conway, Alice: Easy-to-learn three-dimensional scripting for novices, University of Virginia, 1998.

[18] S. G. McLellan, A. W. Roesler, J. T. Tempest, C. I. Spinuzzi, Building more usable APIs, IEEE software 15 (3) (1998) 78–86.

[19] M. Piccioni, C. A. Furia, B. Meyer, An empirical study of API usability, in: Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on, IEEE, 2013, pp. 5–14.

[20] T. R. G. Green, M. Petre, Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, Journal of visual languages and computing 7 (2) (1996) 131–174.

[21] S. Clarke, Describing and measuring API usability with the cognitive dimensions, in: Cognitive Dimensions of Notations 10th Anniversary Workshop, Citeseer, 2005, p. 131.

[22] J. Stylos, S. Clarke, Usability implications of requiring parameters in objects' constructors, in: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 529–539.

[23] J. Stylos, B. A. Myers, The implications of method placement on API learnability, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, ACM, 2008, pp. 105–112.

[24] S. Y. Jeong, Y. Xie, J. Beaton, B. A. Myers, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, D. K. Busse, Improving documentation for eSOA APIs through user studies, in: International Symposium on End User Development, Springer, 2009, pp. 86–105.

[25] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, M. Gordon, What programmers really want: results of a needs assessment for SDK documentation, in: Proceedings of the 20th annual international conference on Computer documentation, ACM, 2002, pp. 133–141.

[26] W. Maalej, M. P. Robillard, Patterns of knowledge in API reference documentation, IEEE Transactions on Software Engineering 39 (9) (2013) 1264–1282.

[27] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, S. R. Klemmer, Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2009, pp. 1589–1598.

[28] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, S. R. Klemmer, Writing code to prototype, ideate, and discover, IEEE software 26 (5) (2009) 18–24.

[29] B. Dorn, A. Stankiewicz, C. Roggi, Lost while searching: Dificulties in information seeking among end-user programmers, in: Proceedings of the 76th ASIS&T Annual Meeting: Beyond the Cloud: Rethinking Information Boundaries, American Society for Information Science, 2013, p. 21.

[30] B. Dorn, M. Guzdial, Learning on the job: characterizing the programming knowledge and learning strategies of web designers, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2010, pp. 703–712.

[31] M. B. Rosson, J. Ballin, H. Nash, Everyday programming: Challenges and opportunities for informal web development, in: Visual Languages

and Human Centric Computing, 2004 IEEE Symposium on, IEEE, 2004, pp. 123–130.

[32] A. Ciborowska, N. A. Kraft, K. Damevski, Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the web, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, ACM, New York, NY, USA, 2018, pp. 94–97. doi:10.1145/3196398.3196467.
URL http://doi.acm.org/10.1145/3196398.3196467

[33] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scafidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, The state of the art in end-user software engineering, ACM Comput. Surv. 43 (3) (2011) 21:1–21:44. doi:10.1145/1922649.1922658.
URL http://doi.acm.org/10.1145/1922649.1922658

[34] P. Pirolli, S. Card, Information foraging., Psychological review 106 (4) (1999) 643.

[35] P. Pirolli, W.-T. Fu, Snif-act: A model of information foraging on the world wide web, in: International Conference on User Modeling, Springer, 2003, pp. 45–54.

[36] B. Athreya, C. Scafidi, Towards aiding within-patch information foraging by end-user programmers, in: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2014, pp. 13–20.

[37] J. Hsieh, M. X. Liu, B. A. Myers, A. Kittur, An exploratory study of web foraging to understand and support programming decisions, in: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2018, pp. 305–306.

[38] S. K. Kuttal, A. Sarma, G. Rothermel, Predator behavior in the wild web world of bugs: An information foraging theory perspective, in: 2013 IEEE Symposium on Visual Languages and Human Centric Computing, IEEE, 2013, pp. 59–66.

[39] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S. D. Fleming, How programmers debug, revisited: An information foraging theory

perspective, IEEE Transactions on Software Engineering 39 (2) (2010) 197–215.

[40] S. K. Kuttal, A. Sarma, M. Burnett, G. Rothermel, I. Koeppe, B. Shepherd, How end-user programmers debug visual web-based programs: An information foraging theory perspective, Journal of Computer Languages 53 (2019) 22–37.

[41] D. Piorkowski, S. D. Fleming, C. Scafidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, A. Horvath, To fix or to learn? how production bias affects developers' information foraging during debugging, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 11–20.

[42] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, C. Swart, Reactive information foraging for evolving goals, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2010, pp. 25–34.

[43] D. Piorkowski, S. Fleming, C. Scafidi, C. Bogart, M. Burnett, B. John, R. Bellamy, C. Swart, Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2012, pp. 1471–1480.

[44] G. Neelima, S. Rodda, Predicting user behavior through sessions using the web log mining, in: 2016 International Conference on Advances in Human Machine Interaction (HMI), IEEE, 2016, pp. 1–5.

[45] C. Bernaschina, M. Brambilla, A. Mauri, E. Umuhoza, A big data analysis framework for model-based web user behavior analytics, in: International Conference on Web Engineering, Springer, 2017, pp. 98–114.

[46] X. Niu, X. Fan, Deep learning of human information foraging behav-ior with a search engine, in: Proceedings of the 2019 ACM SIGIR International Conference on Theory of Information Retrieval, 2019, pp. 185–192.

[47] F. Benevenuto, T. Rodrigues, M. Cha, V. Almeida, Characterizing user behavior in online social networks, in: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, 2009, pp. 49–62.

[48] N. K. Singh, D. S. Tomar, B. N. Roy, An approach to understand the end user behavior through log analysis, International Journal of Computer Applications 5 (11) (2010) 27–34.

[49] R. Saito, T. Kuboyama, Y. Yamakawa, H. Yasuda, Understanding user behavior through summarization of window transition logs, in: International Workshop on Databases in Networked Information Systems, Springer, 2011, pp. 162–178.

[50] S. Otsuka, M. Toyoda, J. Hirai, M. Kitsuregawa, Extracting user behavior by web communities technology on global web logs, in: International Conference on Database and Expert Systems Applications, Springer, 2004, pp. 957–968.

[51] S. Dumais, R. Jeffries, D. M. Russell, D. Tang, J. Teevan, Understanding user behavior through log data and analysis, in: Ways of Knowing in HCI, Springer, 2014, pp. 349–372.

[52] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, I. Kwan, End-user debugging strategies: A sensemaking perspective, ACM Transactions on Computer-Human Interaction (TOCHI) 19 (1) (2012) 1–28.

[53] B. Dervin, An overview of sense-making research: concepts, methods and results to date, INTERNATIONAL COMMUNICATIONS ASSOCIATION ANNUAL MEETING, 1983.

[54] N. K. Agarwal, Making sense of sense-making: tracing the history and development of dervin's sense-making methodology, Int. Perspect. Hist. Inf. Sci. Technol. Proc. ASIS & T (2012) 13.

[55] K. E. Weick, K. M. Sutcliffe, D. Obstfeld, Organizing and the process of sensemaking, Organization science 16 (4) (2005) 409–421.

[56] P. Pirolli, S. C. P. of international conference On, undefined 2005, The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis, phibetaiota.net.
URL https://www.phibetaiota.net/wp-content/uploads/2014/12/Sensemaking-Process

[57] G. Klein, B. Moon, R. R. Hoffman, Making sense of sensemaking 1: Alternative perspectives, IEEE intelligent systems 21 (4) (2006) 70–73.

[58] G. Gao, F. Voichick, M. Ichinco, C. Kelleher, Exploring programmers' api learning processes: Collecting web resources as external memory, in: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020, pp. 1–10. doi:10.1109/VL/HCC50065.2020.9127274.

[59] G. Klein, B. Moon, R. R. Hoffman, Making sense of sensemaking 2: A macrocognitive model, IEEE Intelligent systems 21 (5) (2006) 88–92.

[60] K. Krippendorff, Measuring the reliability of qualitative text analysis data, Quality and quantity 38 (2004) 787–800.

[61] A. J. Ko, B. A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in: Proceedings of the SIGCHI conference on Human factors in computing systems, 2004, pp. 151–158.

[62] J. M. Carroll, M. B. Rosson, Paradox of the active user, in: Interfacing thought: Cognitive aspects of human-computer interaction, 1987, pp. 80–111.